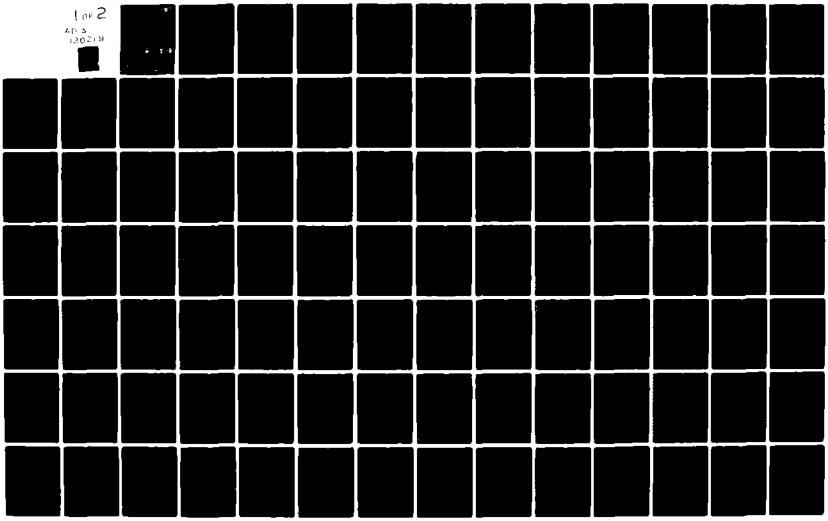
AD-A120 219 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
DAISTS - A SYSTEM FOR USING SPECIFICATIONS AND VERIFICATIONS TO--ETC(U)
JUN 82 P R MCMULLIN F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-82-0867 NL

1 OF 2
AD A
120219



9

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
1807-1856-1920

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

DAISTS
A System for Using Specifications to Test Implementations

by
Paul Raymond McMullin

F49620-80-C-0001

Dissertation submitted to the Faculty of the Graduate School of the University of Maryland in partial fulfillment of the requirements for the degree of Doctor of Philosophy
1982

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 82-0867	2. GOVT ACCESSION NO. AD-A120 219	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DAISTS — A SYSTEM FOR USING SPECIFICATIONS AND VERIFICATIONS TO TEST IMPLEMENTATIONS		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Paul Raymond McMullin		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Directorate of Mathematical & Information Sciences Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 122
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
DAISTS (Data Abstraction Implementation, Specification, and Testing System) combines a data abstraction language containing SIMULA-like classes with algebraic specifications and a library of test monitoring routines. Each axiom equates two sequences of function compositions; the axioms are checked against the implementation by comparing the results of the function compositions for a finite set of test points with a user-supplied equality testing operation. Inconsistencies between the axioms and the implementation are reported as they are detected. While running the tests, the system also monitors (CONTINUED)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONTINUED: statement execution and expression evaluation. By insisting that enough test data be supplied to execute each statement and axiom and to vary the value of each sub-expression, DAISTS helps users develop an effective set of test cases.

The author has evaluated DAISTS both to demonstrate its worth during program development and to find its weaknesses. An experiment was performed that compared program development with DAISTS against more conventional program development. The author showed that 'non-expert' programmers could use formal specifications in program development. The author also used DAISTS in an implementation of a simple text editor, similar to one in Kernighan and Plauger's "Software Toos" book, to show that DAISTS can effectively be employed in the development of complex software.

Abstract

DAISTS (Data Abstraction Implementation, Specification, and Testing System) combines a data abstraction language containing SIMULA-like classes with algebraic specifications and a library of test monitoring routines. Each axiom equates two sequences of function compositions; the axioms are checked against the implementation by comparing the results of the function compositions for a finite set of test points with a user-supplied equality testing operation. Inconsistencies between the axioms and the implementation are reported as they are detected. While running the tests, the system also monitors statement execution and expression evaluation. By insisting that enough test data be supplied to execute each statement and axiom and to vary the value of each sub-expression, DAISTS helps users develop an effective set of test cases.

We have evaluated DAISTS both to demonstrate its worth during program development and to find its weaknesses. An experiment was performed that compared program development with DAISTS against more conventional program development. We showed that "non-expert" programmers could use formal specifications in program development. We also used DAISTS in an implementation of a simple text editor, similar to one in Kernighan and Plauger's "Software Tools" book, to show that DAISTS can effectively be employed in the development of complex software.

For my mother
Ann Boysen McMullin
July 24, 1921 - April 20, 1982

Thanks.

I would like to thank the members of my committee, Dr. Richard Hamlet, Dr. Alan Hevner, Dr. Mark Weiser, and Dr. Marvin Zelkowitz, and my adviser Dr. John Gannon. Dr. Gannon, Dr. Hamlet, and Dr. Mark Ardis helped design the DAISTS system. Support for this project was supplied by the Air Force Office of Scientific Research (AFSOR F49620-80-C-0001) and the University of Maryland Computer Science Center.

Thanks also to Edith Sack for her support and for encouraging me through the dreaded "final semesters".

1. Introduction

Program development remains an error-prone process. Specifications are often ambiguous or incomplete, and validation of a program's conformance to its specification is not easy to accomplish. Program proofs are difficult and expensive to produce, and seemingly minor changes in programs can require extensive modifications to the proofs. Another validation technique, program testing, is often performed by humans who agree too readily with the output of the tested program and who have little feel for how thoroughly the specification or program has been tested. These problems are magnified during the later stages of the software life cycle. Since specifications, programs, and test data are usually separate entities, each can be altered without regard to the others.

We have implemented DAISTS (Data Abstraction Implementation, Specification, and Testing System) which combines a data abstraction language containing SIMULA-like classes with a formal specification and a library of test monitoring routines. Inconsistencies between the specification and the implementation are reported. The system also monitors statement execution and expression evaluation, insisting that enough test data be supplied to execute each statement and to vary the value of each sub-expression. It is this combination of the software technologies of formal specifications, coverage metrics, and high-level implementation languages that makes DAISTS interesting. We have also evaluated DAISTS both to demonstrate its worth during program development and to find its weaknesses.

The next chapter discusses various specification and validation techniques, and their implications and .nits. Chapter three describes our implementation of DAIS[illegible] and the fourth and fifth chapters discuss our evaluat[illegible] of DAISTS: an experiment comparing program developm[illegible] in DAISTS against more conventional strategies, and a case study of the implementation of a simple text editor with DAISTS. The sixth chapter summarizes our experiences with this tool, and offers some suggestions for further research.

2. Background

DAISTS relies on the ideas of specification techniques, testing theory, correctness oracles, and test data completeness criteria. We survey the state of the art in these areas so that the design decisions for DAISTS can be evaluated.

2.1. Specifications

Good specifications are restrictive enough to ensure that no unacceptable implementations satisfy them and general enough so that they preclude no acceptable implementations. They are both concise and plain so that their correspondence to intuitive ideas can be easily judged.

Specifications can be given in a variety of notations that differ in their formality. [Liskov and Zilles 75] review several specification techniques and give criteria for evaluating them. They describe five specification techniques: "fixed discipline," "arbitrary discipline," "state machine model," "axiomatic descriptions," and "algebraic definitions." These techniques fall into two classes: operational techniques (specifying how to implement the objects), and definitional techniques (specifying the properties of the objects without suggesting implementations). We would categorize their "fixed discipline," "arbitrary discipline," and "state machine model" techniques as operational and their "axiomatic" and "algebraic" techniques as definitional.

The following specification fragment is operational because it describes the implementation of the data type _stack_ in terms of _sequences_ of integers.

```
  Pop(S) = if IsEmptySequence(S)
             then S
             else Rest(S)
```

Rest(S) accepts a sequence S and returns a new sequence that is the same as the original except that the first element has been removed. (This definition assumes that the top of the stack is at the front of the sequence.) Any attempt to implement a stack with a different representation will require two conceptual translations: one from the chosen representation into sequences, and another from sequences to stacks. The corresponding fragment of a definitional specification only shows the interaction of the functions of the type, and does not give any information that might predispose an implementor to a specific concrete representation.

```
  Pop(EmptyStack) = EmptyStack
  Pop(Push(S,X)) = if Depth(S) < DepthLimit
                     then S
                     else Pop(S)
```

Operational techniques seem to have the flavor of programming and to suffer most of the same problems: representational dependencies, untreated boundary cases, and arbitrary details such as the order in which conditions are tested. The definitional techniques, because they are less procedural, allow more freedom in choosing the implementation. [Guttag 75] has suggested a method for writing specifications that ensures that boundary cases are explicitly treated.

2.1.1. Using formal specifications

There is little incentive for writing formal specifications unless there is some way to demonstrate the consistency of the specifications and an implementation. Most of the work done in exhibiting this consistency uses the methods of program verification with little or no machine assistance [Hoare 1972], [Wegbreit and Spitzen 1976], [Wulf et al. 1976]. The "pure algebraic" approach [ADJ 1978], [Guttag 1977], [Zilles 1975] avoids assertion-based correctness and implementation. The AFFIRM system [Musser 79] and the OBJ system [Goguen and Tardo 79] provide mechanisms for the symbolic manipulation of algebraic specifications. In the OBJ system, algebraic axioms are used as rewriting rules to "execute" trial expressions, but there is no independent implementation language. AFFIRM joins axiom rewriting with conventional verification of Pascal programs that implement data types. The user interacts with the AFFIRM system to produce verification conditions and guides the system in proving them.

While both systems can be used to prove new properties of specified objects and to exercise specifications before implementation, comparing algebraic specifications to implementations with either system is difficult. The proofs produced with the AFFIRM system require complex assertions to describe the implementation and are valid only for a particular implementation so that a change to the implementation may require a new proof.

2.2. Testing

In contrast to proof methods that attempt to help human beings avoid mistakes, testing methods seek to expose errors. Testing is machine-intensive, and good tests are those that fail--a successful test (i.e., one that exposes no errors) is of dubious value. Path testing [Howden 76], symbolic execution [Clarke 76], domain testing [White and Cohen 80], program mutation [DeMillo, Lipton, and Sayward 78], expression monitoring [Hamlet 77], and dual testing [Panzl 81] have all been suggested as testing strategies. Each testing strategy includes an oracle to determine if the program output is correct for the test input and structural criteria to judge the sufficiency of the test data. Structural criteria improve the confidence obtained from a successful test by excluding the "success" in which a program bug lurks in untested code. The simplest such criteria is that every statement of code be invoked by at least one test point.

2.2.1. Path analysis testing

A path through a program corresponds to a possible flow of control in the program. The subset of the program's domain that causes a particular path to be executed is the path's path domain. The function computed by the sequence of operations in a path is its path computation. The path analysis testing strategy divides the program domain into subsets that correspond to the path domains and generates the set of test data by selecting one element from each non-empty subset. This strategy breaks down when the program contains a loop based on its input variables because the number of paths (and therefore the number of path domains) is potentially infinite.

There are three approximations to path testing to circumvent this problem: level-N path testing, branch testing, and statement testing. Level-N path testing systems only consider paths containing up to N (an arbitrarily fixed number) iterations through the loops comprising the path. Branch testing criteria require that enough test data be supplied so that every branch of a program is executed at least once. Even a program with an infinite number of paths has only a finite number of branches. The completeness of a set of tests for the branch testing criteria does not guarantee correctness, but branch testing completeness is a necessary criteria for correctness in that it is impossible for a set of tests to expose errors in program branches that it does not force to be executed. Early testing systems [Sites 71], [Stucki 73] implemented statement testing by constructing an execution histogram indicating statements that had not been executed. While branch testing requires that test data include values for which while and if statements execute their statement lists zero times, the statement testing criteria is satisfied without such values among the test data. For example, when testing the following loop:

```
 while I > 0 do
      X := (X + I) / (I - 5)
      I := I - 1
   end
```

the statement testing criteria is satisfied with the single test point (I=2). Branch testing would require an additional test point that would execute the loop body zero times, but would be satisfied with the two test points (I=0) and (I=2). Level 2 path testing needs the test points (I=0), (I=1), and (I=2) to execute the loop body zero times, once and twice respectively. These criteria fail to reveal the problem with division by zero when I has the value 5. The error would be exposed by path testing whenever the iteration threshold is greater than 4.

2.2.2. Symbolic execution

Symbolic execution systems [Clarke 76], [King 76] are similar to path testing systems. A path is selected (either automatically or interactively by the user), and the system symbolically executes the program along the chosen path by collecting symbolic representations of the output variables and path constraints expressed in terms of the input variables. By examining the union of these constraint expressions, the user is expected to determine if the path's computation is correct, and if the path constraints are the ones that were expected. Some systems attempt to "solve" or reduce the path constraints and use these solutions to generate test data for the paths, proceeding as in the path testing strategy. These systems can also be used to generate test data that meet criteria other than program correctness: that array indexing expressions stay in bounds, that variables will vary with the input, that division by zero is avoided, and that all branches in a program are feasible.

For example, let us consider the sample program [Clarke 76]:

```
  read A,B
  X := A * B + 2
  if X > 100 then X := 100 - X
  X := X - 50
  if X < 0 then X := 0
  print X
```

If the user specifies the four paths "true true", "true false", "false true", and "false false" (a path being determined by the value of the conditional expressions of the _if_ statements), symbolic execution (with simplified constraints) produces:

Path	Constraints	Result
true true	a*b>98	x = 0
true false	a*b>98, a*b<=48	infeasible
false true	a*b<48	x = 0
false false	a*b<=98, a*b>=48	x = a*b-48

Symbolic execution systems have the same problems with loops that path testing systems have. Additionally, the symbolic expressions produced could be so complex that it may not be easy for the programmer to decide if the path computations are correct.

2.2.3. Domain testing

[White and Cohen 80] presented a testing strategy that detected errors in the control flow of a program by examining the program's performance on test points near the boundaries of its path domains. Their _domain testing strategy_ selected two points from each path domain boundary, and a third point near the domain boundary but outside the domain being examined. They concluded that if the program executed correctly for all three points, then the location of the boundary must have been correct (or at least off by no more than the distance from the boundary to the third point). Their implementation of this technique was limited to programs containing no arrays, and whose path predicates were all linear. Furthermore, it required the program domain to be continuous, and was vulnerable to the problems of <u>coincidental correctness</u> (when a specific data point follows an incorrect path, yet coincidentally computes the correct value), and infinite numbers of path domains (for programs containing input-variable-bounded loops).

2.2.4. Mutation systems

[DeMillo, Lipton, and Sayward 78] found that complex errors in programs were often coupled to simpler ones and that systematic searches for common simple errors could often uncover complex ones. Their testing strategy mutates the original program (the one being tested) by a single token and then runs both the original program and the mutant on the same test data. If the mutant program yields the same computations as the original, then either the mutation performed was a trivial one (e.g., the programs are equivalent), or the test data was not sensitive enough to detect the mutation. A mutant is considered "dead" if it contains detectable errors and "live" if the test data does not distinguish it from the original. Test data that leaves no live mutants (or only mutants that are equivalent to the original) are considered adequate in that either (1) the program is correct or (2) it contains an unexpected error which, by the coupling effect, can be expected to occur infrequently if the mutants are carefully constructed. The mutation testing strategy requires the user to provide enough test data to expose commonly-made errors.

By way of example, consider the following program and one of its mutants:

```
        Original                        Mutant

  subroutine Max(A,N,R)        subroutine Max(A,N,R)
  integer A(N),I,N,R           integer A(N),I,N,R
  R=1                          R=1
  do 3 I= 2,N,1                do 3 I= 2,N,1
3 if(A(I).gt.A(R))R=I        3 if(I.gt.A(R))R=I
  return                       return
  end                          end
```

In the mutant, the reference to "A(I)" has been replaced with "I". When tested with the test data (N=3, A(1)=0, A(2)=-4, A(3)=3), the mutant produces the same value as the original program. In fact, any test data that has the largest value in the array A stored in its last cell will fail to discriminate between the programs; any other test data will expose the error in the mutation. Considered by itself, testing this mutant has little merit, but generating all the mutations in which one variable reference has been replaced by another and finding test data that eliminates them will catch many variable-reference errors in the original program.

This testing strategy was used to develop a set of test data that left only 19 live mutants out of over 22,000 mutations of a 27-statement FORTRAN routine that found the largest integer in an array. The remaining live mutants were shown to be equivalent to the original (correct) routine.

2.2.5. Compiler-based testing

[Hamlet 77] implemented a system that allowed a user to give input-output pairs for specifications of functions and used these pairs as data points to test the functions. As the system tested the behavior of the program (against the input-output pairs), it recorded the history of all the values in the program. After execution, the system examined the expression history and tried all shorter expressions to see if the test data justified the complexity of each expression in the program. If an equivalent expression could be found that was "shorter" than the one in the program's text or if the expression stayed constant, then his system reported the shorter expression, and the user (was to have) supplied more input-output pairs that would justify the program's textual complexity. His system was limited to testing integer functions of single integer parameters.

2.2.6. Dual Testing

Writing two implementations for the same problem, and using each to check the output of the other on randomly selected points from the program domain, has also been suggested [Panzl 81], [Weyuker 80]. Panzl's Dual test driver generates random test cases under the control of a formal specification of the problem's input domain, supplies the test cases to each program, and compares the outputs of the programs. However, any system based on testing on random points from the program domain is unlikely to test boundary conditions, where many errors occur [Goodenough and Gehart 75].

2.2.7. Oracles

Even if a set of test data meets any of the structural criteria described above, the programmer must still determine if the program is producing the correct results on the test data. The process by which a judgement is rendered on the correctness of test results is often call an oracle. Two "oracles" are available: the programmer and the specification. Most testing systems rely on some variation of the programmer as the oracle -- either by having the programmer pre-compute the output value for each input test point or by providing the programmer with some representation of the results and asking him to check that the desired function was computed. In practice, this human oracle is frequently too willing to agree with the program. The results of a computation, even presented symbolically, can be too complicated for a human to judge reliably [Weyuker 80].

Specification-based oracles, such as input-output pairs, are less easily satisfied but limited, encoding only a part of a program's behavior. Testing based on pure input-output pairs has little to recommend it; since any test is finite, it could be met by a program written as a case statement covering the given pairs and otherwise entering a never-ending loop. Another drawback to input-output pairs is that the output must be calculated and represented in a concrete form. Writing input-output pairs for structured objects requires both knowledge of their representation and considerable hand simulation (e.g., to express the result of inserting a new identifier into a hash-coded symbol table).

2.3. Testing theory

While a successful test that reveals no errors increases our confidence in the program, it is of little theoretical value. [Howden 76] provides us with the following theorems:

Theorem: Suppose that P is a program for computing a function F with domain D. There exists a finite subset T of D which can be used to determine the correctness of P, i.e. there exists a finite set T <a subset of> D such that:

P(x) = F(x) for all x in T => P(x) = F(x) for all x in D.

Theorem: There exists no computable procedure H which, given an arbitrary program P and function F with domain D, can be used to generate a nonempty finite set T <a subset of> D such that:

P(x) = F(x) for all x in T => P(x) = F(x) for all x in D.

The first theorem tells us that if a program is not correct, then there must be some element of the program's domain that shows the program is not correct. The second theorem tells us that there is no general procedure for finding those elements that show the program is incorrect. Thus, it is not possible to design an automatic system that can guarantee detection of all of the errors in any program. However, while we know that no general procedure exists, the theory does not rule out our being able to manufacture a system that could "reliably" find the elements that can be used to demonstrate specific errors. Again, [Howden 76] provides these definitions:

Definition: Suppose P is a program for computing a function F whose domain is the set D. Let T be a subset of D. T is a _reliable_ _test_ _set_ for P if:

P(x) = F(x) for all x in T => P(x) = F(x) for all x in D.

Definition: Suppose P is a program, and that H is a strategy for generating subsets T1, T2, ..., Tn of D such that any set T which can be constructed by taking one element from each Ti is a reliable test set for P. Then H is a _reliable_ _test_ _strategy_ for P.

Briefly, that a test set is reliable means that if the program works for all of the elements in the test set, then it will work on any element of the program's domain. This can be achieved in some cases by including all of the program's domain in the test set (exhaustive testing). A routine to sum the initial two cards in a blackjack hand could be tested on all 2652 combinations of cards. Other reliable test sets are significantly smaller than the entire program domain. A reliable test strategy is one that always builds reliable test sets. Again, one obviously reliable test strategy for programs with finite domain is to generate the testset containing all of the program's domain (by putting each element of the domain into its own subset Ti, and then choosing (the) one element from each Ti).

2.4. Evaluating testing strategies

Making use of this theory in the evaluation of testing systems is difficult. When Howden examined a set of 11 programs with well known errors that had appeared in [Kernighan and Plauger 74], he reported [Howden 76] that "path testing was found to be reliable or almost reliable for about 65 percent of the errors." Symbolic execution was found to result in a 10-20% increase in reliability of error detection over similar testing done on actual data values in [Howden 77]. Experimental evidence on the development of 4 programs with Panzl's DUAL testing system has shown that it increased development costs 77%, but drastically reduced the residual errors in programs from 69 to 2.

3. DAISTS

We have built a system called DAISTS (Data Abstraction Implementation, Specification, and Testing System) which combines a high-level implementation language, an independent formal specification notation, and some test data coverage measures. The implementation language allows a programmer to define types via a class construct similar to that of SIMULA 67 [Dahl et al. 70]. DAISTS transforms algebraic specifications [Guttag 77] into test drivers for the implementation. The specifications and a user-defined concrete equality function provide an oracle for judging the correctness of values returned by the functions of the implementation on user inputs. Test data coverage criteria require that each statement in the program be executed and each expression take on multiple values for a successful test to be completed. Two implementations of DAISTS currently exist: one for the UNIVAC 1100 and the other for the VAX 11/780.

The skeletal structure of a DAISTS module is shown below:

```
        .
        . implementation module
        .
    axioms
        .
        . algebraic axioms describing the module
        .
    testpoints
        .
        . declaration and initialization of test values
```

.

<u>testsets</u>

.

. sets of test points to be used with each axiom

.

<u>start</u>

3.1. Implementation language

The implementation language of DAISTS is SIMPL-D [Gannon and Rosenberg 1979], a member of the SIMPL family [Basili 1976] with features that permit the declaration of abstract data types. This language has only modest facilities compared to CLU [Liskov and Zilles 1974] or Mesa [Geschke et al. 1977]; there are no features that support exception handling or non-determinism. Furthermore, the interaction of implementations and axioms in DAISTS forces further restrictions on the use of the implementation language (e.g., side effects are prohibited).

SIMPL-D class declarations define new types that may subsequently be used in object declarations. The interior of a class is a series of variable declarations (the representation) followed by a series of procedure declarations (the body). Including the names of procedures in the class heading makes the procedures visible outside the class. The appearance of the reserved word <u>assign</u> in the operation list enables the assignment operation (:=) to be applied to objects of this class type. As with primitive-type objects, applying the assignment operation to class objects results in the value of the right operand being copied to the location of the left operand.

When a unit of scope containing the declaration of a class object is executed, a new copy of the object is allocated and initialized. Users generally view class objects as indivisible entities; their components cannot be accessed outside the class declaration. Inside the class, these objects may be viewed as structures containing more primitive objects. Statements in the body of a class can access the components of a class object using a period notation similar to that of PL/I or Pascal.

We present here a module containing the definition of the data type "bounded" Stack. The operations available on Stack objects are listed in the class heading; the code implementing these operations (except for the system-defined assignment operation) is found in the class body. An array of EltType (e.g., integer) values and an index of the last value, StackTop, form the representation for Stack objects. Each time a Stack object is bound to storage, space is reserved for the array and integer index. "Push" constructs a new stack object by placing a new value on top of a copy of its input parameter and returns the modified copied object without altering its parameters. Attempts to push a new value on a full Stack object are ignored. "Pop" removes a value from a nonempty Stack object and has no effect on empty Stack objects. "Top" returns the top value of a Stack object, without altering it. "Top" returns Undefined (in the example 0) if it is applied to an empty Stack object. "NewStack" returns as its value an empty Stack object. "Depth" can be used to discover the number of values in a Stack object and "DepthLimit" gives the upper bound on the number of values that can be placed in any Stack object. "StackEqual" judges two stack objects to be equal if they contain the same number of elements, stored in the same order. A definition of "StackEqual" must be included in the implementation because DAISTS invokes it to judge the equality of Stack objects.

```
class Stack = Push, Pop, Top, Empty, NewStack,
               StackEqual, Depth, DepthLimit, assign

  /* macro definitions to control representation */
  define EltType = 'int'
  define Undefined = '0'
  define StackSize = '20'

  /* representation */
  unique EltType array Values(StackSize)
            /* Remember that this array is 0..StackSize-1 */
  unique int StackTop

  /* body containing operation definitions */

  Stack func NewStack
     Stack Result /* local variable of type stack */
     Result.StackTop := -1
     return(Result)

  Stack func Push(Stack S, EltType Elt)
     Stack Result
     if S.StackTop + 1 = StackSize
        then
           return(S) /* return stack unchanged */
        end
     Result := S /* copy to avoid side effects */
     Result.StackTop := Result.StackTop + 1
     Result.Values(Result.StackTop) := Elt
```

```
      return(Result)

   Stack func Pop(Stack S)
      Stack Result
      if Empty(S)
         then
            return(S)
         end
      Result := S /* copy to avoid side effects */
      Result.StackTop := Result.StackTop - 1
      return(Result)

   EltType func Top(Stack S)
      if Empty(S)
         then
            return(Undefined)
         end
      return(S.Values(S.StackTop))

   Bool func Empty(Stack S)
      return(S.StackTop = -1)

   Bool func StackEqual(Stack P, Stack Q)
      int I
      if Depth(P) = Depth(Q)
         then
            I := P.StackTop
            while I >= 0 do /* compare all elements */
               if P.Values(I) <> Q.Values(I)
                  then
                     return(False)
                  end
               I := I - 1
               end
            return(True)
         end
      return(False)

   int func Depth(Stack S)
         return(S.StackTop + 1)

   int func DepthLimit
         return(StackSize)

endclass /* end of declaration of class Stack */
```

3.2. Specifications

Since we are interested in comparing the consistency of specifications and implementations mechanically, a formal specification technique is essential. If the specification is operational, it is likely to be similar to the implementing code. We have therefore chosen a version of the algebraic-axiom technique, whose character is nonprocedural. Thus even though the same programmer might write both specification and code, it is unlikely that one will borrow much from the other. This independence makes the task of comparison more difficult but increases the significance of a successful consistency check.

The primitives of the specification language include boolean and integer constants, free variables, equality operators, other boolean and integer operators, and functional composition. Axioms equate two expressions; the first expression is generally a function composition, and the second expression is a combination of primitives and conditional expressions like those of ALGOL 60. Each axiom presented to the DAISTS processor is named and has a list of the names and types of the free variables used in the axiom. A familiar axiom of bounded stacks appears as:

```
Pop2(Stack S, EltType I):
   Pop(Push(S,I)) = if Depth(S) = DepthLimit
                    then Pop(S)
                    else S;
```

This axiom tells us that the result of popping a Stack S after pushing a new value I onto S is the same as either the value of the object obtained by popping S, or the value of the original Stack S otherwise. (If S already contained the maximum number of values before the push, attempts to push a new value on S would be ignored.)

A set of axioms for bounded stacks appears below:

```
Empty1:
    Empty(NewStack) = True;

Empty2(Stack S, EltType I):
    Empty(Push(S,I)) = False;

Top1:
    Top(NewStack) = Undefined;

Top2(Stack S, EltType I):
    Top(Push(S,I)) = if Depth(S) = DepthLimit
                     then Top(S)
                     else I;

Pop1:
    Pop(NewStack) = NewStack;

Pop2(Stack S, EltType I):
    Pop(Push(S,I)) = if Depth(S) = DepthLimit
                     then Pop(S)
                     else S;

Depth1:
    Depth(NewStack) = 0;

Depth2(Stack S, EltType I):
    Depth(Push(S,I)) = if Depth(S) = DepthLimit
                       then Depth(S)
                       else Depth(S) + 1;

Equal0:
     StackEqual(NewStack,NewStack) = True;

Equal1(Stack P, Stack Q, EltType I):
     StackEqual(P,Push(Q,I)) =
          if StackEqual(P,NewStack)
             then False
             else
          if Depth(Q) < DepthLimit
             then
                if Top(P) = I
                   then StackEqual(Pop(P),Q)
                   else False
             else StackEqual(P,Q);
```

3.3. Supplying Test Data

The testpoints section looks like a procedure, complete with declarations and executable statements. This section allows users to build objects to be referenced in the subsequent testsets section of the program. An object that is expensive to construct can thus be used in testing several axioms without repeating its construction.

Construction of an initial set of test data for bounded stacks is straightforward. The obvious test points are selected: NewStack (an empty stack), Part (a partially full stack), and Full (a full stack). Objects containing these values are constructed below.

```
testpoints /* to be used in the test sets */

   Stack Part, /* a partially-full stack */
         Full  /* a full stack */
   int I

   Part := Push(Push(Push(NewStack,1),2),3)
   Full := Part
   I := 4
   while I <= DepthLimit do
      Full := Push(Full,I)
      I := I + 1
      end
```

The testsets section of the program contains a list of axiom names with values to be substituted for the free variables of the axioms. Test sets need not be given for axioms without free variables (e.g., Empty1, Top1, Pop1, and Depth1); the calls to test them are generated automatically.

The tests for our bounded stack example appear below.

```
testsets /* to "test" the axioms */
  Empty2: (Part,7), (NewStack,1);

  Top2, Pop2, Depth2: /* Note how these share test data. */
          (NewStack,1), (Part,2), (Full,3),
          (Push(Pop(Full),32),4);

  Equal1: (Part,Part,8), (Push(Part,2),Part,2),
          (NewStack,Part,3), (Full,Full,4),
          (Push(Push(Push(Push(NewStack,2),2),3),2),Part,2),
          (Part,NewStack,3),
          (Push(Pop(Full),8),Push(Pop(Full),7),6);
```

3.4. DAISTS´ oracle

The axioms and equality functions serve as DAISTS´ oracle; the oracle is unlimited in that any test case presented can be judged. The tests are turned into calls on the axioms, which act as driver programs for the implementation functions. The left and right sides of an axiom return values that are judged for equality by an appropriate equality function (e.g., StackEqual for bounded stacks). Users must supply an equality function for any types they define. No system default equality (such as component-by-component or bit identity) can capture common tricks of implementation; most abstract types have several concrete representations for the same abstract item. If the result of the equality test is false, a diagnostic message is printed indicating that the axiom has failed for that particular test case.

3.5. DAISTS' structural coverage criteria

DAISTS requires that each part of both axioms and implementing code be executed. Axioms have no "statements," but each conditional expression can be treated as if it were a conditional statement. DAISTS also monitors an expression-analog of executing each statement by isolating each subexpression in both axioms and code. It monitors each expression evaluation during testing and reports any expressions that were not evaluated or that always evaluated to the same value. As in conventional programs an unevaluated expression can result from a compiler optimization of parts of conditional statements (remember, DAISTS is working with production compiled code). An expression that does not vary probably represents a blunder in selecting test data. For example, in the expression:

```
X*(X-1)*(X-5)*(Y+Z)
```

a collection of inputs in which X is always one of {0, 1, 5} will fail to bring out the programmer's need for the final factor. A constant expression within an axiom can reveal a more interesting situation: a failure of the implementation to distinguish abstract objects.

The "debugging" process in DAISTS amounts to obtaining consistent axioms and implementations for the "obvious" test sets and augmenting these with new test data chosen to satisfy the structural criteria. For example, the first four test sets for Equall were chosen to satisfy the four subcases distinguished by the two conditional expressions of the axiom, the next test case hit the only uncovered statement in the StackEqual function (the return(False) in the while loop), and the last two cases were added to make expressions vary.

The test data of the above example satisfies DAISTS. The implementation returns values for both sides of each axiom that are indistinguishable (as integers for EltType and as Stack objects according to StackEqual) for each test set. Each path through the axioms and the statements in the implementing code are exercised, and all relevant subexpressions in both acquire at least two different values. (Not all subexpressions are meant to change value; the Result subexpression returned by NewStack is always the same value.)

3.6. Implementation details

In this section, we describe the compilation process for the portion of the "bounded stack" example shown below.

axioms

```
   Pop2(BoundedStack S, int I):
      Pop(Push(S,I)) =
          if Depth(S) = DepthLimit
            then Pop(S)
            else S;
```

```
testpoints

   BoundedStack S1,S2 /* "local" variables */
   S1 := Push(Push(Push(NewStack,1),1),2)
   S2 := S1
   while Depth(S2) < DepthLimit
     do
       Push(S2,Top(S2)+Top(Pop(S2)))
     end

testsets

   Pop2: (NewStack,7), (S1,22), (S2,83),
         (Push(Pop(S2),19),82);

start
```

The scanner essentially modifies the above text (inserting calls to the run-time library, and compiler directives as needed) to look like:

```
proc Pop2(BoundedStack S, int I,
          int Testsetnumber)

/+ monitoring on +/
     if StackEqual(Pop(Push(S,I)),
                   exprif Depth(S) = DepthLimit
                     exprthen Pop(S)
                     exprelse S)
       then
          return /* axiom held true */
       endif
/+ monitoring off +/
     call Report_axiom_failure('Pop2',Testsetnumber)

proc main testdriver

     BoundedStack S1,S2

     /* initialize the library so that we can safely
        use the implementation */
     call monitoring_initialization

     S1 := Push(Push(Push(NewStack,1),1),2)
     S2 := S1
     while Depth(S2) <> DepthLimit
       do
          S2 := Push(S2,Top(S2)+Top(Pop(S2)))
       end
```

```
    /* re-initialize library discarding any incidental
       monitoring information gathered while initializing
       S1 and S2 */
    call monitoring_initialization

    call Pop2(NewStack,7,1) /* testset 1 (parm 3) */
    call Pop2(S1,22,2) /* testset 2 */
    call Pop2(S2,83,3) /* testset 3 */
    call Pop2(Push(Pop(S2),19),82,4) /* testset 4 */

    /* testing is finished, report test results */
    call monitoring_report

start maintestdriver
```

The /+ monitoring ... +/ directives are produced so that the parser does not invoke the monitoring library for code that is generated to call the "axioms" (in maintestdriver) or code generated to report axiom failures. The "Pop2" axiom is expanded into calls on implementation routines that compute the values of the left and right sides of the axiom, and "StackEqual" compares these values. If "StackEqual" returns false, an error message is generated reporting the failure of the axiom to hold for specific test data. The testsets section has been translated into a series of calls on the procedure that implements the "Pop2" axiom.

The parser translates this "text" into quads, adding quads to monitor expression evaluation and branch execution. The quads that are produced for the procedure "Pop2" are similar to:

```
   1   Segment Pop2
   2   Call StackEqual              result=t1 type=boolean
   3   Call Pop                     result=t2 type=stack

/* The quads for the sub-expression "Push(S,I)".
 * Note that the parameters are listed after the call quad.
 */

   4   Call Push                    result=t3 type=stack
   5   expression_monitor S         expr_number=1
   6   parm S
   7   expression_monitor I         expr_number=2
   8   parm I
   9   endparms

   10  parm t3
   11  expression_monitor t3        expr_number=3
   12  endparms
   13  parm t2
   14  expression_monitor t2        expr_number=4

/* quads for the expression if (right side of axiom) */

   15  Call Depth                   result=t5 type=integer
   16  expression_monitor S         expr_number=5
   17  parm S
   18  endparms
   19  expression_monitor t5        expr_number=6
   20  call DepthLimit              result=t6 type=integer
   21  endparms
   22  = t5 t6                      result=t7 type=boolean
   23  expression_monitor t7        expr_number=7
   24  exprif t7                    result=t4 type=stack

/* Quads for the "then Pop(S)" half of the if expression */

   25  call Pop                     result=t5 type=stack
   26  expression_monitor S         expr_number=8
   27  parm S
   28  endparms
   29  expression_monitor t5        expr_number=9
   30  exprthen t5                  result=t4 type=stack

   31  expression_monitor S         expr_number=10
   32  exprelse S                   result=t4 type=stack

/* the if expression is a parameter to "StackEqual" */

   33  parm t4
   34  endparms
   35  if t1
```

```
36  then
37  return
38  endif
39  call report_axiom_failure
40  parm ´Pop´
41  parm Testsetnumber
42  endparms
```

To ensure that each expression takes on two different values, expression evaluation is monitored at run time. Each monitored expression is assigned a unique expression number, but some expressions (t1, t4, and t6) are not monitored. StackEqual (t1) should always be true (or the "Pop2" axiom will fail to hold for a test case and an error message will be generated). The if expression (t4) is not monitored because its value is either the value of its "then" part (t5 - line 29) or its "else" part (S - line 31), both of which must have varying values to satisfy expression monitoring. The call to DepthLimit (t6, line 20) is also not monitored because the function has no parameters and cannot have side effects; the value it returns cannot vary and is treated as a constant. This type of analysis leads us to avoid monitoring expressions like:

```
   Push(Push(NewStack,24),4*20)
```

because "NewStack", 24, 4, and 20 are all "constants".

For each monitored expression, the code generator produces a call to a run-time library routine whose arguments are the address of the expression to be monitored, its number, its size, and the address of an equality function whose arguments are the same type as the expression.

The run-time expression-monitoring routine saves a copy of the value of the expression being monitored (if there is room in a storage pool) the first time that the expression is evaluated. On all subsequent evaluations, the routine compares the copy against the (current) value of the expression. Once an expression has changed value, its initial value can be discarded. A global flag, "notmonitoring", inhibits monitoring of "equal_routine" when it is invoked by "monitor_expression". The structure of the run-time expression monitor is as follows.

```
rec proc monitor_expression(number,address,
                                  size,equal_routine)

  if notmonitoring or alreadyvaried(number)
    then
      return
    end

  if alreadymonitored(number)
    then /* a value from the first evaluation is
            * in the storage pool. See if it changed */
      notmonitoring := true /* don't monitor equal_routine
                               * when called from system
                               * routine */
      if not equal_routine(oldvalue(number)->value,
                               address->value)
        then /* value has changed! record this fact! */
          freespace(oldvalue(number)) /* return storage */
          alreadyvaried(number) := true
        end
      notmonitoring := false /* re-enable monitoring */
      return
    end

  /* haven't saved an old value yet! */
  if not space_available(size)
    then /* not enough storage space to save value now! */
      seen_not_saved(number) := true
      return
    end

  /* there is room to save the value! */
  save_value(size,address,old_value(number))
  already_monitored(number) := true

end  /* monitor_expression */
```

Since the UNIVAC implementation did not have a large address space, the storage pool for saving values was a statically allocated buffer of about five thousand words, which generally was sufficient for testing modules up to approximately 400 lines in length (200 lines of abstract type implementation, 150 lines of specification, and 50 lines of test data). On occasion the storage pool overflowed and some of the expression values could not be saved on their first evaluations. In these cases, DAISTS reports the list of expressions for which data had been lost. The programmer may then run the testing session again (either with a larger static storage pool or with extra test data so that some of the expressions whose values did not vary on the first testing session would vary, freeing up some of the storage pool). The VAX implementation is able to make use of the VAX's large address space for storage allocation to save the expression values dynamically.

A problem with any system that monitors program execution is that of reporting the results to the programmer. It is easy to map statement numbers to source text line numbers. However, there is no simple way to map expression numbers onto text lines. DAISTS provides a listing of expression numbers with the text of the expressions they represent by traversing the quads at compile time (via an inorder traversal, treating identifiers as "leaves" in the tree of quads), and inserting parenthesis as required by operator precedences. The output for our example looks similar to:

Expr number	Text Line	Expression
1	3	S
2	3	I
3	3	Push(S,I)
4	3	Pop(Push(S,I))
5	4	S
6	4	Depth(S)
7	4	Depth(S) = DepthLimit
8	5	S
9	5	Pop(S)
10	6	S

Sample monitoring messages appear below.

```
Expression number 7 always evaluated to the same value.

Expression numbers 8, 9 were never evaluated.
```

4. Experiment

We performed an exploratory study that compared program development with DAISTS against a more conventional programming technique. The goals of the study were to show (1) that the use of DAISTS would reduce the number of delivered errors, (2) that using DAISTS would not prolong the program testing process, aids often produce insufficient tests.

In evaluating DAISTS there are really two primary issues to be resolved: 1) the ease with which users write axioms, and 2) the ability of users to develop programs consistent with axioms. We concentrated on the second issue, program development, rather than specification development. Obviously, if the program development task proved too difficult, the system's worth would be questioned because writing formal specifications before development would only make the process more difficult.

4.1. Methodology

An intermediate class in programming languages was divided into two groups, given identical English language descriptions of the abstract type "bounded-list-of-integers", and assigned to produce implementations of twelve operations for the type in SIMPL-D. The axiom group used algebraic specifications to test their implementations, while the control group used the more traditional test driver method. The axiom group was given the axioms for the type, and the control group was given a test driver that used most of the abstract operations to sort groups of integers. Subjects were allowed to produce other axioms or test drivers at their discretion. Since some control group subjects would undoubtedly test with only the sort routine while others would write their own test drivers (at least for the three functions not used by the sort routine), we would get some information about their ability to judge test coverage. Both groups had to develop their own test data. At the end of the experiment, we examined the programs to discover the residual errors.

4.1.1. Choosing the groups

The class (of 79 students) met together for lecture twice a week, and was divided into four smaller groups that met once a week with one of two teaching assistants. Two of the small groups (one from each assistant) were combined to form the axiom group (45 students), and the other two groups formed the control group (34 students). Five control group subjects and four axiom group subjects did not turn in any project, and were dropped from the study. Despite warning the subjects several times that every compilation was being recorded and that they were required to work independently, three pairs of projects were so (remarkably) similar that we could not objectively consider them to be independent efforts. One of each pair of the "non-independent" projects (the "dependent" one if determinable) has been omitted. This left 40 subjects in the axiom group, and 27 subjects in the control group.

To check that the two groups were of approximately the same ability, we analyzed the grades that they received for the semester. This analysis showed only one statistically significant difference between the two groups -- the control group had slightly higher examination scores. (See Table 1 below.)

Table 1 Group Differences.

		Axiom Group	Control Group
Letter grade	Mean	2.44	2.54
(A = 4.0)	St dev	.90	.94
	Level	< 80%	
Project grade	Mean	120.8	114.1
(Max = 150)	St dev	17.8	26.8
	Level	< 44%	
Exam Grade	Mean	41.00	45.81
(Max = 100)	St dev	12.12	8.90
	Level	< 7%	

Letter grades on a scale 1=D, 2=C, etc.
Project grades (five projects not counting experiment) on a 150 point scale.
Composite exam grades on a 100 point scale.

Significance levels for a two-tailed Mann-Whitney U-test [Siegel 1956].

4.1.2. The Project

We chose to assign a small project (approximately 150 lines) to implement "bounded list-of-integers." The operations were described in English in the project handout (see Appendix I), which also contained instructions for using the appropriate processor. The subjects were told that they were to implement all of the functions described in the handout and present results demonstrating that the sort routine or axioms executed with no obvious errors.

A manual describing the implementation language and giving other specific information (how the axiom group accessed the axioms and how the control group could obtain the standard driver or write their own drivers) was also distributed. The axioms and the test driver program that were provided are in Appendices II and III respectively.

Since the test data had to be submitted along with the program at compile time for the axiom group (to allow DAISTS to construct its test driver), the control group was also required to submit their test data with their compilation requests to make the development environments more nearly equivalent.

We also provided separate lecture and laboratory meetings for the two groups, and tried to exchange experimenters so that each group met with each experimenter to nullify any bias in our lectures. The subjects were informed that they had been divided into two groups, and were asked not to exchange information about how the two groups were different. However, some of the details of the SIMPL-D language were discussed with both groups present.

4.1.3. Data Collection

A special processor was set up to limit access to DAISTS; it prohibited members of the axiom group from writing separate test drivers and members of the control group from using the axioms. This processor also saved a copy of a every deck submitted to allow us to examine program development histories. This approach led to several identical decks being saved by the submission processor -- either a subject would run a deck at his terminal and then run the deck again to generate a listing on the printer, or programs failed to complete execution before their system-default time limit was exhausted so the decks were resubmitted with larger time limits.

4.1.4. Identifying errors

After the subjects' projects were collected and graded, the files of decks saved by the submission processor were examined. For each subject, the deck that corresponded to the listing that was submitted by the subject for grading was separated into the class definition and the debugging data. We then debugged each implementation, both by "desk checking" and by using the DAISTS system. Many of the errors that we found were detected simply by turning on the subscript-checking feature of the compiler (apparently few of the subjects used this feature).

As we were debugging implementations, we found that several implementations had "subjective restrictions" that were not addressed in the project assignment. These restrictions could also be interpreted as errors, but a case could be made for allowing them. For example, one subject stored only single-digit positive integers. Also, several subjects could correctly store any integers except zero, which they used as markers in their representations.

When these "subjective errors" were included, the results were not substantially different from the results reported below, which come from only counting the "objective errors" -- code that fails no matter how favorable the input values selected.

4.1.5. Measuring errors

We also faced a dilemma in choosing how to report errors. We feel that the most conservative objective measure that we can use is functions containing errors. If a function in the submitted project had to be changed, regardless of the number of changes that had to be made to correct the function, it was counted as a single "function containing an error." We like the resolution of this measure because 1) it is more nearly representation-independent than any other error measure; and 2) the project description defined functions, the axioms specified functions, and the sort routine used the specified functions.

We compared our measure to distinct errors and error occurrences [Gannon 77]; where, if the same error in computing the length of a list was made in three places, it would count as one distinct error, but three error occurrences. Our data produced similar results for both of these measures and for the measure "functions containing errors."

Several of the subjects made errors in selecting their representations. One subject used a circular list and had an ambiguity in his representation between a full list and an empty one. This error could only be fixed by adding a word to his representation and repairing many of the functions. Another subject's project was corrected by merely changing the size of an array in his representation (no functions needed changing). These subjects were charged with one incorrect function to account for the change to the representation (in addition to any incorrect functions for which they were charged).

In the results reported below, the measure <u>functions that contained objective errors</u> was used.

4.1.6. Measuring Cost of Development

It is inherently difficult to measure programmer effort. It is especially difficult to measure the effort of students who do not work regular schedules and who are not inclined to keep track of effort on a project near the end of a semester. We used <u>number of distinct runs</u> as the measure of development cost since it was the only enforceable metric that we could employ.

4.2. Results

Our hypotheses that the axiom group would deliver fewer errors than the control group is tested by comparing both the number of functions that contained errors 1) counting the functions that were assigned, and 2) counting only the functions contained in the sort routine given to the members of the control group. Our hypotheses that DAISTS usage would not increase the program development cost is tested by comparing the number of distinct runs used by the members of each group. The hypotheses that the control group members could not judge the effectiveness of their test coverage is tested two ways: 1) by looking for a difference between the control group's performances on just the functions tested by the sort routine and their performances on all the functions, and 2) by considering the performance of the sub-groups of the control group that (a) did, and (b) did not write their own test driving routines.

Since the sort program tested 9 of the 12 list operations, we have reported both the number of incorrect functions tested by the sort routine and the total number of incorrect functions. The subjects were required to implement all the functions and were encouraged to write extra functions to display list objects as a debugging aid.

In the tables below, we report the means and standard deviations of the number of incorrect functions tested by both the axioms and the sort routine, the total number of incorrect functions, and the number of distinct runs. The significance levels shown were generated by one-tailed, Mann-Whitney U-tests [Siegel 56].

4.2.1. Original Groups

The two groups had similar numbers of incorrect functions when we consider only those functions tested by the sort routine, and the number of distinct runs were also similar. While the means favored the members of the axiom group (an average of .18 fewer incorrect functions out of the 9 functions tested by the sort routine and .71 fewer distinct runs), the differences were not significant. As expected, the axiom group did significantly better than the control group in eliminating errors in all the functions assigned.

Table 2 All Axiom (40) and Control (27) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.60(1.18)	.78(1.64)	NS
All incorrect functions	.82(1.51)	1.78(2.20)	<.003%
Distinct runs	11.77(5.65)	12.48(8.53)	NS

4.2.2. Successful Subjects

We found that some of the subjects turned in projects that were not "successfully completed" -- subjects in the axiom group had messages about inconsistent axioms and implementations, and subjects in the control group turned in projects for which the sort routine would not correctly sort <u>their</u> test data. By successfully completing the project, we mean only that the output of the program that was submitted for grading displayed no obvious errors. In the axiom group, 32 of 40 subjects successfully completed the project, and 22 of 27 control group subjects were successful.

Considering only those subjects who successfully completed the project, the axiom group did marginally better than the control group even on the functions tested by the sort routine. This result appears despite the sort routine doing a good job of exposing the errors in these routines for those subjects choosing good sets of data to use with it. (When the data for the sort program contained all the boundary cases of the sort routine, all the boundary cases of the list functions were tested). Of course, the results are even more striking when we consider all the functions. The axiom group delivered more correct functions than did the control group without taking more runs.

Table 3 Successful Axiom (32) and Control (22) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.12(.33)	.23(.42)	<20%
All incorrect functions	.19(.39)	1.23(1.20)	<.004%
Distinct runs	10.97(5.60)	11.41(7.60)	NS

We have subdivided the successful control group into two groups for further comparison: those that wrote and ran small test driver programs in addition to running the sort program (which is more like an integration test than a unit test), and those that used the sort routine exclusively for testing. Of the 22 successful control group members, 7 wrote their own driver programs.

4.2.3. Subjects Writing Their Own Drivers

We expected that those subjects in the control group who wrote driver programs would test as effectively as the axiom group, but would require more runs to debug their own drivers. The data in Table 4 supports these hypotheses for the sort functions only. Even driver-writing subjects did not produce as many correct functions as the axiom group did.

Table 4 Successful Axiom (32) and Driver-Writing (7) Subjects

	Axiom group	Driver group	Level
Incorrect sort functions	.12(.33)	.14(.35)	NS
All incorrect functions	.19(.39)	.57(.49)	<2%
Distinct runs	10.97(5.60)	15.14(5.82)	<4%

Examining the runs of the driver-writing subjects to determine why their efforts did not match those of the axiom group, we find a distinct lack of testing discipline. Five of the 7 subjects had test drivers that could exercise all the functions. Four of the subjects used effective tests, trying a variety of objects in different operations, while two other subjects with extensive test drivers just did not seem to use enough data to cover the necessary cases. Four of the subjects used drivers before using the sort routine seriously as an integration test. Two other subjects used drivers only in response to specific errors that occurred in testing with the sort routine.

4.2.4. Subjects Testing with the Sort Program Only

Those subjects testing only with the sort program used an average of 1.3 fewer runs than did the members of the axiom group. However, they did not produce as many working functions, even when we consider only the functions tested by the sort program (8.73 to 8.88). Part of the explanation for this result may be that the sort program did not encourage the members of the control group to test more thoroughly. The sort program's effectiveness as a testing vehicle was impaired by poor selections of test data that did not include the boundary cases of the sort's domain.

Table 5 Successful Axiom (32) and Sort-Only (15) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.12(.33)	.27(.44)	<14%
All incorrect functions	.19(.39)	1.53(1.31)	<.002%
Distinct runs	10.97(5.60)	9.67(7.70)	<8%

4.3. Conclusions from the Experiment

While the axiom group needed slightly more runs to satisfy DAISTS, its members correctly developed more functions. The discipline of testing with DAISTS helped even inexperienced users avoid less systematic testing methods. Even if we consider only the subjects in our study who wrote their own test drivers, we observe a variety of questionable testing practices - omitted functions, failures to consider boundary cases, and generally insufficient test data. The formal specification required by DAISTS identifies the boundary cases and clearly defines their treatment. Furthermore, DAISTS run-time monitoring routines ensure that the code handling boundary cases is exercised.

Performing this study gave us insights that could help us improve our system. Several residual errors might have been exposed by special-values testing strategies [Howden 78], e.g., adding tests that include the zeros of the types (0 for integers, null and blank strings, NewList, etc.) and the constants that appear in the text of the implementation ('AddFirst(NewList,22)', '76', '"constant strings"', etc.).

5. Use of DAISTS in an Implementation

While the experiment reported above attempts to show that DAISTS can be used to reduce errors in the program development process, it does not address the problem of using specifications and tools on larger software projects. Only a few reports of their use for other than small examples have appeared in the literature: [Gerhart and Wile 79] specified and verified part of a multiple user file updating module using algebraic axioms with the AFFIRM system; [Guttag and Horning 80] specified a display interface. This chapter relates our experience using DAISTS to specify and implement a record-oriented text editor similar to one described in [Kernighan and Plauger 81].

5.1. Design Overview

The text editor was implemented in five modules including data type definitions for records, files, and patterns, and modules to implement a record-oriented editor and a simple user interface. The module-interface structure of the system is shown below:

```
                User
             Interface
                  |   \----------
                  v    \         \
               Editor   \        |
                / | \    \       |
          -----   |  ---  \      |
         /        |     \  |     |
        v         |      v v     |
  Patterns        |      Files   |
     |            |        |     |
     \            v        |    /
      ------> Records <---------
```

The editor implementation takes two files and interprets one file as a list of editing commands to be applied to the other "text" file. Currently, the user interface contains only a small loop that reads a record from the terminal, converts it to a file (containing a single record), submits it to the editor with the text file (built by previous commands), and saves the result in the text file. This trivial user interface could easily be replaced with a more powerful one that would include "meta-commands" for accessing permanent disk files and reading command streams.

5.1.1. Basic Editor Commands

The editor module updates its text file and maintains a pointer to the current record in the file. Editor commands are line-oriented and operate relative to the current position in the file. Each command may have an optional address (or address range) that indicates the file position (or range of file positions) where it is applied. The basic editor commands are shown below.

i<text>	Inserts the <text> into the file after the current record, leaving the editor positioned at the inserted record. If the file is positioned before its first record, the inserted record becomes its new first record.
d	Deletes the current record, positioning the editor to the previous record. If there is no previous record, the editor is positioned before the first record.

p (or <Nullrecord>)	Prints the current record, without changing the position in the file.
s<D><pattern><D><text><D>	Substitutes <text> for the first occurrence of the <pattern> in the current record. The delimiter <D> may be any character not appearing in the <pattern> or the <text>. The position of the editor is not changed. If the <pattern> is not present in the current record, no change is made.

5.1.2. Command Addresses

Prefixing a command with an address positions the editor at the record selected by the address and then executes the command. Prefixing a command with an address range (indicated by two addresses separated by a comma) causes the command to be executed at each of the records indicated by the range and positions the editor at the record selected by the second address. If the second address of a range identifies a file position that is before that identified by the first address, it is ignored. After the last execution of an insert, delete, or substitute command, the editor prints the (then current) record.

Addresses in the editor may be indicated in any of the following ways:

<number>	The <number>'th record in the text file. If <number> is zero or negative, the position is BEFORE the first record of the file. If <number> is greater than the length of the file, the position is that of the LAST record of the file.

.	The current position of the editor (at the time that the command is interpreted).
$	The last record of the file.
/<pattern>/	The file is searched, starting at the record after the current one, toward the end of the file, until a record containing the <pattern> is found. The position of the record containing the <pattern> is the address used. If the <pattern> is not present in the file after the current record, the address of the last record in the file is used.
\<pattern>\	The file is searched, starting at the record before the current one, toward the beginning of the file, until a record containing the <pattern> is found. The position of the record containing the <pattern> is the address used. If the <pattern> is not present in the file before the current record, the address zero (before the first record of the file) is used.

5.1.3. Patterns

Patterns are used to identify file positions and to select parts of records for replacement in the substitute command. The patterns that we implemented are similar to those of [Kernighan and Plauger 81]:

c	literal character
?	any character
^	beginning of record
$	end of record
[...]	character class (match any of these characters)
[^...]	negated character class (all but these characters)
*	closure (zero or more occurrences of previous pattern)
@c	escaped character (e.g. @[, @^, ...)

Character classes contain one or more of the following elements:

c	literal character, including [
a-c	range of characters (digits, lower or upper case)
@c	escaped character (@^, @], @@, ...)

Patterns match the leftmost sequence of characters in a record, matching as many characters as possible. For example, the pattern "a*bc*" matched with the record "xbzaaabccccc" matches one character in the second character position of the record, and matched with the record "xyzaaabccccc" matches nine characters starting at the fourth character position.

5.1.4. Sample Editing Commands

Some sample editing commands are shown below.

1,.d	deletes all of the records from the beginning of the file up to (and including) the current record. The editor will be positioned before the (then) first record of the file.
.,$p	prints all of the records from the current one to the end of the file, leaving the editor positioned at the last record of the file.
\begin\,/end/s/x/y/	changes the first occurrence of x to y on each of the records from the most recent "begin" to the next "end". The editor is positioned at the record containing "end" (or at the last record of the file if "end" is not present).
/[A-Z][A-Za-z0-9]*/	prints the next record of the file containing an identifier starting with a capital letter, leaving the editor positioned there. If no record contains such an identifier, the last line of the file is printed.

5.2. Specifying the Types

Formal specifications were written for four modules: records, files, patterns, and editor. We did not specify the user interface because communicating with the environment is accomplished via side effects.

5.2.1. Records

The type record presents no specification problems because it behaves much like the (familiar) type "character string" found in programming languages like PL/I. Functions generate null records, measure record lengths, convert a character into a record of length one, extract the first character from a record, delete the first character of a record, concatenate two records, extract a subrecord of a record, locate a (shorter) record in a (longer) one, and judge the equality of records. To support files and patterns, we later added functions to locate an unescaped character in a record, convert a record to an integer, and to maintain a sorted set of characters in a record.

5.2.2. Files

Our type files is more like that of [Cleaveland 80] than the files of Pascal. Records are inserted into, or deleted from, a file at the position of the file's read/write head. Inserting a record causes the head to be positioned at the new record, while deleting a record causes the head to be positioned at the preceding record. Other operations move the head forward and backward in the file, test if the file is positioned before the first record or at the last record, return the address or value of the current record (i.e., the one under the head), and replace the current record.

One of the most interesting axioms for files defines the composition of Advance and Insert.

```
Advance(Insert(R,X)) =
   if Atend(X)
     then Insert(R,X)
     else Insert(CurrentRec(Advance(X)),
                Replace(R,Advance(X)));
```

Part of the reason that this axiom was difficult to write is that the Advance operation modifies only the positioning of the file, while the Insert operation modifies the contents and the positioning of the file.

If the file is already positioned at the last record, Advance has no effect. Otherwise, the value of the record that should wind up under the read/write head is extracted (by "CurrentRec(Advance(X))"), the file is advanced, the record that was to have been inserted replaces the extracted value (via "Replace(R,Advance(X))"), and the extracted value is reinserted, making it the (now) current record in the file.

In [Cleaveland 80] the axiom specifying the result of advancing past an inserted record was:

```
Advance(M(X,Insert(R,Y))) = M(Insert(R,X),Y)
```

where M maps two files (the first containing all records up to and including the current record and the second containing the remaining records in reverse order) into a single file. This has the effect of using the second file as a stack of records, and an obvious implementation would be to implement a file as two stacks of records. Although nobody would implement a file this way, these specifications have more representational bias than ours.

Taking the file apart and reconstructing it in the axioms allowed us to avoid a hidden function to keep track of the position of the read/write head. We were able to use CurrentRec to "hold" the value of a record, perform the appropriate manipulation, and then re-insert the "held" value. For example, in the specification of Delete(Back(X)), two records are deleted and the value of the first deleted record is re-inserted.

```
  Delete(Back(X)) =
       if Attop(Back(X))
         then Back(X)
         else Back(Insert(CurrentRec(X),Delete(Delete(X))));
```

5.2.3. Patterns

Our implementation of the pattern matcher closely follows that in [Kernighan and Plauger 81] except that we did not allow patterns to match across record boundaries.

Our specification for patterns employs a technique used by [Gerhart and Wile 79] in the Delta experiment. They built sequences of instructions describing the net effects of operations on tree-like files by interpreting the words representing the "constructor histories" of files, and then applied the instructions to a fresh copy of the file to obtain its new value. Two of our functions, PatfromRec and MatchPat, performed similar tasks. PatfromRec interpreted records containing patterns into sequences of pattern-construction operations producing an object of type pattern. Then MatchPat used the pattern objects to determine the position within a record where the pattern matched.

The PatfromRec specification is shown below.

```
PatfromRec(R) =
  if RecordEqual(R,NullRec)
    then NullPat
  else if CharofRec(R) = "@"
         then /* escaped character */
           if RecLength(R) = 1
             then ErrorPat /* no character present */
             else ConcatPat(Lit(CharofRec(Rest(R))),
                            PatfromRec(SubRec(R,3,MaxLen)))
  else if CharofRec(R) = "?"
         then /* Arb */
           ConcatPat(Arb,PatfromRec(Rest(R)))
  else if CharofRec(R) = "*"
         then /* Closure */
           ConcatPat(Closure,PatfromRec(Rest(R)))
  else if CharofRec(R) = "^"
         then /* Beginning of Line */
           ConcatPat(Bol,PatfromRec(Rest(R)))
  else if CharofRec(R) = "$"
         then /* End of Line */
           ConcatPat(Eol,PatfromRec(Rest(R)))
  else if CharofRec(R) = "["
         then /* Choice */
           ConcatPat
             (Choice(SubRec(R,1,EscapedFind(R,"]","@"))),
              PatfromRec(SubRec(R,EscapedFind(R,"]","@")+1,
                                  MaxLen)))
  else /* literal character */
         ConcatPat(Lit(CharofRec(R)),PatFromRec(Rest(R)));
```

For example, a record containing the characters:

^X*$

to match a line consisting of zero or more X's is interpreted by PatfromRec into the sequence of pattern operations:

```
ConcatPat(Bol,     /* beginning of line "^" */
         ConcatPat(Lit('X'), /* literal "X" */
                  ConcatPat(Closure, /* "*" */
                           /* and end of line "$" */
                           ConcatPat(Eol,NullPat))))
```

Either a single pattern within a record (e.g., missing characters as in '[a-z' and '@', ambiguous range specifications like '[f-3]' and '[m-d]'), or the combination of patterns within a record (e.g., beginning of line patterns not at the beginning of the pattern like 'a^', extra patterns after an end of line pattern like '$a*', and multiple closures like 'a**') can be incorrect. PatfromRec detects single pattern errors while translating a record into a concrete pattern object; and MatchPat detects errors arising from the contextual use of a pattern while pattern matching. We made this distinction because we found that PatfromRec did not (in general) distribute with respect to concatenation, i.e.:

```
  PatfromRec(R1 cat R2) <> PatfromRec(R1) cat PatfromRec(R2)
```

For example, "@" is the escape character and "?" is Arb (a pattern that matches any single character), but

```
  PatfromRec(ConcatRec("@","?")) =
             PatfromRec('@?') = Lit("?")
```

yields a pattern to match the literal character "?", while

```
  ConcatPat(PatfromRec("@"),PatfromRec("?")) =
             ConcatPat(ErrorPat,Arb) = ErrorPat
```

produces the error pattern.

MatchPat determines the position within a record where a pattern matches. The fragment of the MatchPat definition shown below reveals that pattern matching (for patterns containing closures) and contextual error checking (for beginning-of-line patterns within larger patterns) is performed by looking ahead at the next pattern in P. For example, if the first subpattern of P is an Arb, MatchPat looks ahead at the next piece of the pattern:

```
  FirstPat(RestofPat(P))              for Arb
  FirstPat(RestofPat(RestofPat(P)))   for Arb* and fails the
```

match (returns 0) if a beginning-of-line pattern is found. If this were not done, MatchPat would fail to recognize that the internal beginning-of-line pattern was not the first subpattern of P because of the recursive style of the definition.

```
MatchPat(R,P) =
  if IsNullPat(P)
   then 1 /* a Null pattern always matches */
  else if IsErrorPat(P)
        then 0 /* error patterns never match */
  else if IsEol(FirstPat(P))
        then /* End of Line */
          ...
  else if IsBol(FirstPat(P))
        then /* Beginning of Line */
          ...
  else if IsArb(FirstPat(P))
        then /* Arb matches any character */
         if IsClosure(FirstPat(RestofPat(P)))
          then /* Arb* */
           if IsBol(FirstPat(RestofPat(RestofPat(P))))
            then
             0 /* interior Bol, no match possible */
            else
             if IsNull(R) /* null record */
```

```
                    .or. MatchPat(Rest(R),P) = 0
                          /* pattern doesn't match record */
                   then /* Arb* matches null record */
                     /* record with rest of pattern? */
                     MatchPat(R,RestofPat(RestofPat(P)))
                   else 1 /* Arb* starts match here */
           else /* Arb */
             if IsBol(FirstPat(RestofPat(P))) .or.
                                 /* interior Bol? */
                IsNull(R) .or. /* null record */
                MatchPat(Rest(R),RestofPat(P)) = 0
                                 /* rest of pattern doesn't
                                  * match rest of record */
               then 0 /* no match */
               else /* Arb matches First(R) only
                          * if rest of match OK */
                 MatchPat(Rest(R),RestofPat(P))
  ...
```

The MatchPat axiom is unusual in that its left side is not a composition of operations on records and patterns. Instead of breaking down operations by composing them on the left side of the axiom, the MatchPat definition recursively decomposes the pattern on the right side of the axiom using FirstPat and RestofPat. We would normally have tried to write an axiom like:

```
MatchPat(R,ConcatPat(P1,P2)) =
  if MatchPat(SubRec(R,MatchPat(R,P1)
                    +MatchLen(R,P1),1000),P2) = 1
    then MatchPat(R,P1)
    else ?
```

to indicate that if the second part of the pattern matched the part of the record after that matched by the first part, the result is the point at which the first pattern matched. However, the term:

```
  MatchPat("aa",ConcatPat("a*","a")))
```

presents difficulty since "a*" matches the entire record and none of the record remains to match the final "a". A recursive definition could be written to handle this problem by specifying right-to-left pattern matching while reducing the amount of the record matched by the first pattern until a match is found for the second pattern. However, even this scheme is not satisfactory since the term:

```
  MatchPat("abcaba",ConcatPat("ab","a"))
```

would match the first "ab" and the last "a" instead of the second "ab" and the last "a".

Two implementation biases occur in our specification of pattern matching operations. First, the ValofChoice function converts a (concrete representation of a) choice pattern into a record containing a list of characters for later use. The axiom relating ValofChoice and Choice is representationally biased in that it specifies that the list of resulting characters is sorted (via SortedCharSet) and that the underlying representation of alphabetic characters is contiguous.

Second, we require the characters in a choice pattern to be sorted so that the equality operations for patterns can be specified without additional hidden functions. The concrete representations of choice patterns are converted to sorted records of characters by ValofChoice and choice-pattern equality can be reduced to record equality. An alternate approach would have used ConcatRec to build an unordered record containing the characters in the choice pattern and to introduce a hidden function, AllAreIn, to check that all the characters of its first argument were contained in its second argument. Given two choice patterns C1 and C2, the axioms describing these operations would be:

```
PatternEqual(C1,C2) = /* C1 and C2 are Choice patterns */
     AllAreIn(ValofChoice(C1),ValofChoice(C2)) .and.
     AllAreIn(ValofChoice(C2),ValofChoice(C1));

AllAreIn(R1,R2) = /* R1 and R2 are records */
     if IsNull(R2)
       then true /* every char in R2 is in R1! */
       else MatchRec(R1,First(R2)) <> 0 .and.
            AllAreIn(R1,Rest(R2));
```
where MatchRec returns the location of its second argument in its first.

5.2.4. Editor

The editor module was not implemented as a type with a concrete representation as the other modules were, but as a module providing operations on files of records. Its axioms describe how to interpret records as editor commands to be applied to a file of text. Since we did not anticipate using the editor module functions in any other module, we felt less restraint about the number of hidden functions that we used in its specification. These hidden functions had two applications: parsing the commands (i.e., extracting an address field and basic command from a record), and applying them to the text file (i.e., positioning the text file and applying the basic command throughout an address range).

We were surprised with the number of times that we used the underlying types to specify and implement the editor functions (e.g. the pattern matching functions were useful in extracting addresses from commands). This differs from our usual experience of building new types that merely manipulate the underlying types without using their operations to any real advantage.

5.3. Implementation Experience

The following table shows the sizes of the different pieces of the system.

module name	code lines	axioms lines	testdata lines	functions count	axioms count
editor	241	220	75	9	8
files	254	145	68	18	34
patterns	877	881	181	29	143
records	186	164	57	19	26
totals	1558	1310	381	75	211

Although the number of lines of axioms and the number of axioms looks forbidding, many axioms were trivial. The pattern module had 9 primitive functions (basic pattern types) and eight functions for discriminating among them. Thus 72 of the 143 axioms looked like:

```
Is_Primitive_Pattern1(Primitive_Pattern2) = false
```

5.3.1. Specification Problems

We chose to implement the text editor from [Kernighan and Plauger 81] because it was a well-known, carefully documented problem. Nevertheless, we encountered several difficulties with the informal documentation. No association rule was given for ranges, so the pattern [a-c-e] could be interpreted as [abce@-], [acde@-], or even [a-e]. Also their informal documentation did not explain when the editor's "current position" is moved during command interpretation. Consider the editor command:

```
  \begin\,/end/p
```

and the file:

```
  begin
     write('This write contains the word: end')
     /* other text here */
     x := 1 /* "current position" of the editor here */
  end
```

If the fourth line is the "current position" of the editor, the forward search for the "end" pattern could start from the "current position" or from the position found by the backward search for the pattern "begin". Producing formal specifications made us consider these cases.

5.3.2. Unit Testing

During unit testing, we ensured that the axioms and implementations were consistent for our test data. In addition, DAISTS' run-time monitors insisted that we supply enough test data so that every axiom branch and implementation statement was executed, and that each expression in the axioms and code had at least two different values during testing. Expression coverage at the 95% level is easy to achieve with common test data, but each additional percentage point of coverage required great effort.

There were a few expressions in tests that we could not get to vary. These expressions fell into one of two categories: either they were tests to ensure safe failing when the input arguments were outside the normal function domain, or they were truly redundant.

An example of the safe-failing test occurred when we included a check in the pattern matching routine to see that the pattern argument was initialized. Since our pattern variables were always initialized, the outcome of this test never varied.

Code improvements are sometimes foregone because the worst-case analysis of algorithms makes it impossible to tell if a certain improvement is allowable. A compiler-based tool can judge both a program and its test data, and insist that no improvement is possible [Hamlet 77a]. DAISTS detects these improvements as expressions that do not vary or statements that are not executed. For example, the MatchLen function in the pattern module was constructed by copying the code for the MatchPat function, and then modifying it to return the number of characters matched rather than the position of the match. Since MatchLen uses MatchPat to do the "recursive lookahead" necessary for the pattern matching, a number of places in MatchLen could be optimized; they were hold-overs from the MatchPat function and could no longer occur. It is not clear whether to delete these holdovers in the name of efficiency, or to maintain the parallel construction for ease of maintenance.

5.3.3. Integration Testing

We found only two errors during integration testing. The routine to extract subrecords returned a subrecord with an extra character when the start and length of the subrecord equaled the length of the original record. The original >= in the following code had to be changed to >.

```
Record func SubRec(Record X, int First, int Len)
  ...
  if First + Len >= Length(X.Val)
    then
      Result.Val := X.Val[First]
    else
      Result.Val := X.Val[First,Len]
    end
  return(Result)
```

The error would have been detected if we had used the testing criteria in [Howden 81] in which arithmetic relation functions, E1 r E2, must be evaluated over values for which E1<E2, E1=E2, and E1>E2.

The other integration error occurred in the routine that maintained sorted sets of characters, which inserted duplicate characters. This error would not have been found with any unit testing strategy because both the specification and the implementation for the function were wrong (in the same way). It was not until the module was integrated with the patterns module that the error was detected. After we realized that the wrong function had been implemented, DAISTS was still useful in testing the corrected specification and implementation. It is interesting to note that this error occurred in a function introduced to support the implementation bias discussed above; the function itself did not provide an interesting operation.

5.4. Conclusions from the Case Study

The use of DAISTS in this case study must be considered successful -- only two errors escaped unit testing. There was only one instance where the orthogonality of DAISTS´ (applicative) specification language and its (imperative) implementation language allowed us to produce a specification and an implementation that were both wrong, and one other case where the implementation and the specification disagreed but escaped exposure by our simple unit testing criteria.

As other researchers have reported, producing formal specifications reveals (interesting) boundary conditions that are often omitted in even the most carefully constructed informal problem statements. Using the specifications with a testing tool forced our implementations to handle these boundary conditions.

The use of algebraic axioms as a formal specification language was unwieldy for several of the functions that we implemented. Our specifications contain many "recursive definitions" that do not look like typical algebraic axioms. The tests in these definitions are ordered, making them more like applicative programs than algebraic axioms that distribute their tests among several independent axioms. Giving up the independence of axioms appears to reduce the orthogonality between the specification and the implementation. [Gerhart and Wile 79] report similar experiences with the axioms in the Delta system.

6. Conclusion

6.1. Advantages

We feel DAISTS has several advantages over conventional program development systems:

(1) The specification, program, and test data are packaged as a single entity, encouraging their mutual maintenance.

(2) The specification language is applicative and the implementation language is imperative. This orthogonality reduces the likelihood of the same error appearing in both the specification and the implementation.

(3) The test data coverage of the specification and the program are measured. Thus it is unlikely that the specification makes distinctions that the implementation does not, and vice versa.

(4) There is no need to describe the concrete representation of the results of an operation in the specification; the user (specifies and) writes an equality routine to judge the results of tests for abstract objects. This simplifies the testing process by removing the requirement for "hand simulation" of complicated operations.

(5) Having a tool that incorporates specifications into the development process should provide the motivation and experience necessary for programmers to use formal specifications effectively.

When measured by the number of persistent errors in the development of our text editor, DAISTS´s use must be considered successful -- only two errors were found during integration testing. The orthogonality of DAISTS´ specification language and its implementation language resulted in only one error for which both the specification and the implementation were wrong. We were also pleased by the degree to which our coverage measures complemented the formal specifications: there was only one integration error in which the implementation and the specification disagreed despite passing both hurdles.

Our classroom experiment showed that DAISTS can help users formulate more effective tests. Axioms are also effective for exploring boundary conditions, where errors frequently occur [Goodenough and Gerhart 75]. DAISTS uses the simplest structural test criteria imaginable, but it is surprising how well they direct testing. When a programmer has written the code and a set of axioms, and chosen intuitively appealing test sets, the module usually fails the structural coverage requirements (often in the axioms because of boundary cases). Data added to achieve coverage may then expose an inconsistency (often that the boundary cases were omitted from the code). Furthermore, even for the simplest modules it is impossible to second-guess DAISTS' responses: the bookkeeping required to see that a set of tests succeeds, including the structural requirements, can only be done accurately by a computer. The subjects of our study who used only the sort program to test their implementations stopped testing too soon because the data they fed the sort program did not expose errors in their implementations. Even the subjects in our study who wrote their own test drivers exhibited a variety of questionable testing practices - omitted functions, failures to consider boundary cases, and generally insufficient test data.

We were frustrated by the imprecision inherent in informal specifications. In our experiment, subjects stretched our careful English descriptions to restrict their implementation to handle only single-digit integers and to disallow a value that was used as an internal marker. We experienced similar ambiguities with the informal specifications in [Kernighan and Plauger 81]; no association rule was given for ranges and no explanation was given for when the editor is moved during command interpretation. DAISTS' formal specifications usually avoid this imprecision by forcing explicit enumeration and treatment of boundary cases. We only noticed the ambiguity of the text editor descriptions when we wrote our algebraic specifications.

6.2. Drawbacks

Despite these advantages, DAISTS is not without flaws. If there are no failures, DAISTS reports that test sets were tried without incident. As in any testing, this "success" is of doubtful significance. The implementation and axioms could contain compensating errors; also, implementation or axioms alone could be wrong yet hide the error by internal compensation. It is our belief that implementations and axioms are unlikely to compensate for each other's errors because of their orthogonal natures. It is more likely that one description of the data type contains canceling errors.

Equality function implementations are the most likely to contain compensating errors. The ultimate such error is an "equality" which always returns true (ignoring differences between the objects it was to have compared). Then all axioms apparently succeed on any data, no matter what the other function implementations do (so long as they return values). A programmer is not likely to trivialize the implemented type in this way because he must understand the equality interpretation to understand the representation function. Furthermore, the structural test requirements often catch the error of making too many things equal. For example, none of the expressions of type Stack could vary if

Stackequal always returns true because DAISTS uses this function to determine when an expression's value has varied. However, DAISTS must admit the possibility of a bad equality implementation. Consider the following functions purporting to implement "Pop", "Push", "Top", and "StackEqual" for bounded stacks:

```
Stack func Push(Stack S, EltType Elt)
   Stack Result
   if S.StackTop + 1 = StackSize
      then
         return(S) /* return stack unchanged */
      end
   Result.Values(1) := S.Values(0) /* copy current top value */
   Result.StackTop := Result.StackTop + 1
   Result.Values(0) := Elt
   return(Result)

Stack func Pop(Stack S)
   Stack Result
   if Empty(S)
      then
         return(S)
      end
   Result.Values(0) := S.Values(1) /* copy next to top elt */
   Result.StackTop := Result.StackTop - 1
   return(Result)

EltType func Top(Stack S)
   if Empty(S)
      then
         return(Undefined)
      end
   return(S.Values(0))

Bool func StackEqual(Stack P, Stack Q)
   if Depth(P) = Depth(Q)
      then
         if P.Values(0) <> Q.Values(0)
            then
               return(False)
            end
         return(True)
      end
   return(False)
```

The functions have all been adjusted to keep only the current and previous values that were pushed onto the stack. Since none of the axioms push more than one value or pop more than one value, and since the StackEqual axiom only looks at the current top value, no monitoring can detect these compensating errors.

The current version of the system requires that the parameters of a function not be altered by the implementation. Otherwise separate occurrences of free variables in axioms could contain different values. DAISTS also requires that each function mentioned in the specification be implemented; if the specification uses hidden functions, each of them must appear in the implementation for the module to be tested.

In our case study we found algebraic axioms to be an unwieldy specification notation for several of the text editor modules. Problems occurred in modules that were not really data types in which values are stored and retrieved, but collections of related functions (e.g. the editor module). Our specifications contained many "recursive definitions" which did not look like typical algebraic axioms. Recursive definitions appear imperative while typical axioms seem declarative: the order of the tests on the right sides of recursive definitions is fixed, while the algebraic axioms distribute these tests among several independent axioms. [Gerhart and Wile 79] seemed to have had similar experiences.

6.3. Enhancements

DAISTS could profitably be extended in several directions: with the ability to manipulate the specifications directly (so that they could be exercised as rewrite rules), with the inclusion of other formal specification languages (that handle procedures with side effects and communication interfaces), and by the addition of interactive run-time monitors to allow construction of test sets to extend coverage incrementally.

We feel that DAISTS-like systems might expose some additional errors with more stringent structural criteria like special-values testing strategies [Howden 78] (e.g., adding tests that include the constants of the types and the those that appear in the text of the implementation) and functional testing [Howden 81] (e.g., evaluating arithmetic relation functions, E1 r E2, over values for which E1<E2, E1=E2, and E1>E2). This enhanced structural criteria would have exposed residual errors that we discovered in our subjects' programs and in our editor implementation. The desire for such additional measures must be balanced against the difficulty of achieving them. During unit testing of the text editor, we found expression coverage at the 95% level easy to achieve with common test data, but each additional percentage point of increased coverage required great effort.

The computer science community has a growing consensus on the desirability of formal specifications. Writing formal specifications would not be considered an overhead activity if they could be used to reduce the effort of writing and testing implementations. A tool, such as DAISTS, which can incorporate formal specifications into the development process could provide the motivation and experience necessary to produce those specifications.

7. References

[ADJ 78]
J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. in Current Trends in Programming Methodology 4, R. Yeh, ed., Prentice-Hall, Englewood Cliffs, N.J., 1978, 80-149.

[Basili 76]
Victor R. Basili. The Design and Implementation of a Family of Application-oriented Languages. Proceedings of the Fifth Texas Conference on Computing Systems, (October 1976), 6-12.

[Clarke 76]
Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. IEEE Trans. Software Eng. SE-2, no 3, (September 1976), 215-222.

[Cleaveland 80]
J. Craig Cleaveland. Mathematical Specifications. SIGPLAN Notices 15, no 12, (December 1980), 31-42.

[DeMillo, Lipton, and Perlis 79]
Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. Comm. ACM 22, no 5, (May 1979), 271-280.

[DeMillo, Lipton, and Sayward 78]
Richard A. DeMillo, Richard J. Lipton, and Fredrick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer 11, no 4, (April 1978), 34-41.

[Gannon 77]
John D. Gannon. An Experimental Evaluation of Data Type Conventions, Comm. ACM 20, no 8, (August 1977), 584-595.

[Gannon and Rosenberg 79]
John D. Gannon and Jon Rosenberg. Implementing Data Abstraction Features in a Stack-based Language. Software - Practice and Experience 9, no 7, (July 1979), 547-560.

[Gerhart and Wile 79]
Susan L. Gerhart and David S. Wile. Preliminary Report on the Delta Experiment: Specification and Verification of a Multiple-User File Updating Module. Proceedings of Specifications of Reliable Software, (April 1979), 198-211.

[Geschke et al. 79]
C.M. Geschke, J.H. Morris, Jr., and E.H. Satterwaite. Early Experience with Mesa. Comm. ACM 20, no 8, (August 1977), 540-552.

[Goguen and Tardo 79]
Joseph A. Goguen and Joseph J. Tardo. AN INTRODUCTION TO OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. Proceedings of Specifications of Reliable Software, (April 1979), 170-189.

[Goodenough and Gerhart 75]
John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. IEEE Trans. Software Eng. SE-1, no 2, (June 1975), 156-173.

[Guttag 75]
John V. Guttag. The Specification and Application to Programming of Abstract Data Types, (Ph. D. dissertation), Computational Sciences Group, University of Toronto, report no CSRG-59, (September 1975).

[Guttag 77]
John V. Guttag. Abstract Data Types and the Development of Data Structures. Comm. ACM 20, no 6, (June 1977), 396-404.

[Guttag et al. 78]
John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract Data Types and Software Validation. Comm. ACM 21, no 12, (December 1978), 1048-1064.

[Guttag 79]
J.V. Guttag. Notes on Type Abstraction. Proceedings of Specification for Reliable Software, (April 1979), 36-46.

[Guttag and Horning 80]
John Guttag and J.J. Horning. Formal Specification as a Design Tool. Proceedings of the 7th ACM Principles of Programming Languages, (January 1981), 251-261.

[Hamlet 7,a]
Richard G. Hamlet. Testing Programs with Finite Sets of Data. The Computer Journal 20, no 3, (March 1977), 232-237.

[Hamlet 77b]
Richard G. Hamlet. Testing Programs with the Aid of a Compiler. IEEE Trans. Software Eng. SE-3, no 4, (July 1977), 279-290.

[Hamlet 77c]
Richard G. Hamlet. Compile-Time Testing. Proceedings of the Sixth Texas Conference on Computing Systems, (November 1977), 1A-15 - 1A-21.

[Hoare 72]
C.A.R. Hoare. Proof of Correctness of Data Representations. Acta Informatica 1, (1972), 271-281.

[Howden 76]
William E. Howden. Reliability of the Path Analysis Testing Strategy. IEEE Trans. Software Eng. SE-2, no 3, (September 1976), 208-215.

[Howden 77]
William E. Howden. Reliability of symbolic evaluation. Proceedings IEEE COMPSAC 77, (November 1977), 442 - 447.

[Howden 78]
William E. Howden. An Evaluation of the Effectiveness of Symbolic Testing. Software - Practice and Experience 8, no 4, (July-August 1978), 381-397.

[Howden 81]
William E. Howden. Completeness Criteria for Testing Elementary Program Functions. Proceedings of 5th International Conference on Software Engineering, (March 1981), 235-243.

[Kernighan and Plauger 74]
Brian W. Kernighan and P.J. Plauger. The Elements of Programming Style. McGraw-Hil Publishing Company, New York, 1974.

[Kernighan and Plauger 81]
Brian W. Kernighan and P.J. Plauger. Software Tools in Pascal. Addison-Wesley Publishing Company, Reading, MA., 1981.

[King 76]
J. C. King. Symbolic Execution and Program Testing. Comm. ACM 19, no 7, (July 1976), 385-394.

[Liskov and Zilles 75]
Barbara H. Liskov and Stephen N. Zilles. Specification Techniques for Data Abstractions. IEEE Trans. Software Eng. SE-1, no 1, (March 1975), 7-18.

[Musser 79]
David R. Musser. Abstract Data Type Specification in the AFFIRM System. Proceedings of Specifications of Reliable Software, (April 1979), 47-57.

[Panzl 81]
David J. Panzl. A Method for Evaluating Software Development Techniques. Journal of Systems and Software 2, no 2, (June 1981), 133-137.

[Parnas 72]
David L. Parnas. A Technique for Software Module Specification with Examples. Comm. ACM 15, no 5, (May 1972), 330-336.

[Siegel 56]
Sidney Siegel. Nonparametric Statistics for the Behavioral Sciences, McGraw-Hill, New York, 1956.

[Sites 71]
R.L. Sites. ALGOL W Reference Manual. Stanford University, Department of Computer Science, STAN-CS-71-230, (1971).

[Stucki 73]
L.G. Stucki. Automatic generation of self-metric software. Proceedings IEEE Symp. Computer Software Reliability, (1973), 94-100.

[Wegbreit and Spitzen 76]
B. Wegbreit and J. Spitzen. Proving Properties of Complex Data Structures. Jour. ACM 23, no 2, (April 1976), 389-396.

[Weyuker 80]
Elaine J. Weyuker. On Testing Nontestable Programs. Computer Science Department Technical Report, no. 025, New York University, (October 1980).

[White and Cohen 80]
Lee J. White and Edward I Cohen. A Domain Strategy for Computer Program Testing. IEEE Trans. on Software Eng. SE-6, no 3, (May 1980), 247-257.

[Wulf et al. 76]
W.A. Wulf, R.L. London, and M. Shaw. An Introduction to the Construction and Verification of Alphard Programs. IEEE Trans. Software Eng. SE-2, no 4, (December 1976), 253-265.

[Zilles 75]
S.N. Zilles. Algebraic Specification of Data Types. Project MAC Progress Report for 1973-74. (MIT CSG Memo 119), 1-12.

8. Appendix I - Project assignment for the experiment

8.1. First page of assignment

You are to write the CLASS implementation for lists of integers. A list is an ordered collection of elements that may have elements added and deleted at its ends, but not in its middle. The operations that you must "export" are: AddFirst, AddLast, Conc, DeleteFirst, DeleteLast, First, IsEmpty, ListEqual, ListLength, NewList, and Reverse. Each operation is described in detail below.

The lists are to contain up to eleven (11) elements. If an element is added to the front of a "full" list (one containing eleven elements already), the element at the back of the list is to be discarded. Elements to be added to the back of a full list are discarded. Requests to delete elements from empty lists result in empty lists, and requests for the first element of an empty list results in zero (0).

Remember that the operations that you implement are to be functions, and that they may ***NOT*** change their parameters! If a function needs to manipulate a parameter to perform the operation, the parameter is to be COPIED to a LOCAL variable BEFORE the change is performed! You may use any representation you choose to implement your lists. The detailed operation descriptions are below:

List FUNC Addfirst(List L,INT I) - Returns the list with I as its first element followed by the elements of L. If L is "full" to start, L's last element is ignored.

List FUNC Addlast(List L,INT I) - Returns the list with the elements of L followed by I. If L is full to start, I is ignored.

List FUNC Conc(List L1,List L2) - Returns the list made up of the elements of list L1 followed by the elements of L2. If L1 and L2 together contain more than eleven (11) elements, then the extras are to be ignored.

List FUNC Deletefirst(List L) - Returns the list containing all but the first element of L. If L is empty, then it returns an empty list.

List FUNC Deletelast(List L) - Returns the list containing all but the last element of L. If L is empty, then it returns an empty list.

Appendix I - Project assignment for the experiment

INT FUNC First(List L) - Returns the first element in L. If L is empty, then it returns zero (0).

INT FUNC Isempty(List L) - Returns one (1) if L is empty, zero (0) otherwise.

INT FUNC Listequal(List L1,List L2) - Returns one (1) if the two lists are element for element equivalent (e.g. First(L1) = First(L2),...), and zero (0) otherwise. Note that two empty lists are considered equal.

INT FUNC Listlength(List L) - Returns the count of elements in L. An empty list has a count of zero (0) elements.

List FUNC Newlist - Returns a list initialized to be empty

List FUNC Reverse(List L1) - Returns a list containing the elements of L1 in reverse order.

Appendix I - Project assignment for the experiment

8.2. Second page of assignment (control group)

A test routine has been written for you, or you may write your own test routines. The provided routine reads in groups of integers, sorts them, and prints out the smallest 11 of each group. The test routine expects the groups of integers to be separated by zero. A sample test run using the provided test routine is shown below:

```
 @add simpld*project.setup           <done once per run>


 @$.SIMPLD,S                         <calls the compiler,
                                        asks for listing>
 <list implementation>               <your CLASS for lists>
 $TEST                                 <causes   the   test
routine
                                        to be provided>
 <groups of integers, separated by zeros>
 @eof                                <end of the data>
```

The data for the test routine may have any number of integers or groups of integers per card, with the integer 0 separating each group. Spaces are used to separate the integers when more than one integer is on a card.

A sample run f . using your own test routine is shown below:

```
 @$.SIMPLD,S                           <call   compiler   as
above,
                                        assuming  setup   is
done>
 <list implementation>
 <your test driver>
 $DATA
 <your data>
 @eof
```

You will be required to submit your list implementation via the deck submission processor (to be discussed in class), and you will also turn in a listing of a run using the provided test routine, that shows several groups of integers correctly sorted.

8.3. Second page of assignment (axiom group)

Axioms have been written that you must use to debug your CLASS. You may add axioms of your own at your discretion. A sample run is shown below:

```
 @add simpld*project.setup        <done once per run>


 @$.SIMPLD,S                      <calls compiler, asks
                                     for listing>
 <list implementation>            <your CLASS for lists>
 $AXIOMS                           <causes   axioms   to   be
provided>
 <your optional axioms>
 TESTPOINTS
 <your testpoints>
 TESTSETS
 <your testsets>
 START
 @eof                              <that's all you need>
```

You will be required to submit your list implementation via the deck submission processor (to be discussed in class), and you will also turn in a listing of a run using the provided axioms, with no axiom failures and all statements executed.

AD-A120 219 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
DAISTS - A SYSTEM FOR USING SPECIFICATIONS AND VERIFICATIONS TO--ETC(U)
JUN 82 P R MCMULLIN F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-82-0867 NL

2 OF 2
AD A
120219

END
DATE
FILMED
11-82
DTIC

9. Appendix II - The Axioms supplied to the axiom group Axioms

```
/* These axioms are constructed following the Guttag
   rules for deciding which axioms need to be constructed.
   The functions in the "O" group are:
       IsEmpty, ListEqual, ListLength, First
   The functions in the "TOI1" group are:
       NewList, AddFirst
   The functions in the "TOI2" group are:
       AddLast, DeleteLast, DeleteFirst, Conc, Reverse
*/

IsEmpty1:
   IsEmpty(NewList) = 1;

IsEmpty2(List AxList1,int AxInt1):
   IsEmpty(AddFirst(AxList1,AxInt1)) = 0;


ListEqual1:
   ListEqual(NewList,NewList) = 1;

ListEqual2(List AxList1,int AxInt1):
   ListEqual(NewList,AddFirst(AxList1,AxInt1)) = 0;

ListEqual3(List AxList1,int AxInt1):
   ListEqual(AddFirst(AxList1,AxInt1),NewList) = 0;

ListEqual4(List AxList1,List AxList2,int AxInt1,int AxInt2):
   ListEqual(AddFirst(AxList1,AxInt1),AddFirst(AxList2,AxInt2))
=
       if AxInt1 <> AxInt2
         then 0
         else
          if ListLength(AxList1) = 11
            then /* Need to trim the end off! */
             ListEqual(DeleteLast(AxList1),DeleteLast(AxList2))
            else /* Compare them just as they are! */
             ListEqual(AxList1,AxList2);


ListLength1:
   ListLength(NewList) = 0;

ListLength2(List AxList1,int AxInt1):
   ListLength(AddFirst(AxList1,AxInt1)) =
      if ListLength(AxList1) = 11
        then 11
        else 1 + ListLength(AxList1);
```

Appendix II - The Axioms supplied to the axiom

```
Firstl:
   First(NewList) = 0;

First2(List AxListl,int AxIntl):
   First(AddFirst(AxListl,AxIntl)) = AxIntl;


/* Now for the "TOI2" function definitions: */

AddLastl(int AxIntl):
   AddLast(NewList,AxIntl) = AddFirst(NewList,AxIntl);

AddLast2(List AxListl,int AxIntl,int AxInt2):
   AddLast(AddFirst(AxListl,AxIntl),AxInt2) =
      AddFirst(AddLast(AxListl,AxInt2),AxIntl);


DeleteLastl:
   DeleteLast(NewList) = NewList;

DeleteLast2(List AxListl,int AxIntl):
   DeleteLast(AddFirst(AxListl,AxIntl)) =
      if IsEmpty(AxListl)
        then NewList
        else
         if ListLength(AxListl) = 11
           then
AddFirst(DeleteLast(DeleteLast(AxListl)),AxIntl)
           else AddFirst(DeleteLast(AxListl),AxIntl);


DeleteFirstl:
   DeleteFirst(NewList) = NewList;

DeleteFirst2(List AxListl,int AxIntl):
   DeleteFirst(AddFirst(AxListl,AxIntl)) =
      if ListLength(AxListl) = 11
        then DeleteLast(AxListl)
        else AxListl;


Concl(List AxListl):
   Conc(NewList,AxListl) = AxListl;

Conc2(List AxListl,List AxList2,int AxIntl):
   Conc(AddFirst(AxListl,AxIntl),AxList2) =
      AddFirst(Conc(AxListl,AxList2),AxIntl);


Reversel:
   Reverse(NewList) = NewList;
```

Appendix II - The Axioms supplied to the axiom

```
Reverse2(List AxList1,int AxInt1):
   Reverse(AddFirst(AxList1,AxInt1)) =
      if ListLength(AxList1) = 11
        then AddLast(Reverse(DeleteLast(AxList1)),AxInt1)
        else AddLast(Reverse(AxList1),AxInt1);
```

10. Appendix III - The sort routine for the control group

```
proc Main /* The driver for the sort program to test
lists. */
/* Read in a series of numbers that  ends  with  zero,  sort
them, */
/* and then print out the smallest "ListSize" of them. */

int Holder /* A place to read numbers into */
int Setnumber /* The number of the current set */
int Counter /*  A  counter  of  the  number  of  numbers  in
Unsorted */
List Unsorted /* Where the unsorted numbers are read into */
List Sorted   /* Where the sorted numbers are stored! */

/* Main loop for reading in sets of numbers */

Setnumber := 1 /* Start to work on the first set */
while .not. eoi do
     Sorted := NewList /* No sorted numbers in this group */
     Unsorted := NewList /* Also no unsorted numbers in yet!
*/
     read(Holder)             /* To initialize Holder! */
     while Holder <> 0 .and.
          .not. eoi do      /* Keep reading and sorting */
          Counter := 0  /* Set to count the unsorted list */
          while Holder <> 0 .and.
               Counter < ListSize .and. .not. eoi do
               Unsorted := AddFirst(Unsorted,Holder)
               Counter := Counter + 1
               read(Holder) /* Get the next  number  of  the
set */
            end

          /*  Either  Unsorted  is  full,  or  this  set  is
finished! */
          /* Must first join unsorted  numbers  with  sorted
ones */

          Sorted := Merge(Sorted,Sort(Unsorted))
       end

     /* Here we must have hit an end of a set! */
     /* Print out the first "ListSize" worth of numbers */

     write(skip,'Sorted numbers of set number',Setnumber)
     while .not. IsEmpty(Sorted) do
          write(First(Sorted)) /*Output smallest  number  in
set*/
          Sorted := DeleteFirst(Sorted)
       end
```

Appendix III The sort routine for the control group

```
    Setnumber := Setnumber + 1 /* Now start the next set */
  end write(skip,skip,skip,'Out of sets of numbers to sort')


List func Merge(List M1in,List  M2in)  /*Merges  two  sorted
lists*/

List Result
List  M1,M2  /*  Locals  so  that  we  do  not  change   the
parameters! */

M1 := M1in /* Copy parameter */
M2 := M2in /* Copy second parameter too! */
Result := NewList
while .not. (IsEmpty(M1) .or. IsEmpty(M2))
  do /* Done when one is empty! */
     if First(M1) <= First(M2)
       then /* Take next value from M1 */
          Result := AddFirst(Result,First(M1))
          M1 := DeleteFirst(M1) /* Don't need  first  number
*/
       else /* Take from M2! */
          Result := AddFirst(Result,First(M2))
          M2  :=  DeleteFirst(M2)  /*  Discard  first  after
copying */
       end
  end

/*One of  the  two  lists  is  empty  -  catenate  them  all
together!*/

return(Conc(Reverse(Result),Conc(M1,M2)))   /*and    reorder
Result!*/


rec List func  Sort(List  Inlist)  /*Sorts  into  increasing
order*/

/* This procedure works by a merge sort - Split the list  in
*/
/* two, sort each half, and then merge the two sorted halfs!
*/

List Half1,Half2

Half1 := NewList /* Initialize! */
Half2 := Inlist /* Initialize! */
while ListLength(Half1) < ListLength(Half2) do
     Half1 := AddFirst(Half1,First(Half2))
```

Appendix III The sort routine for the control group

```
   Half2 := DeleteFirst(Half2)
end return(Merge(Sort(Half1),Sort(Half2)))
```

11. Specifications from the Case Study

```
define true = '1', false = '0', NullChar = 'charval(0)'
```

11.1. Axioms forRecords

```
RecLength0:
  RecLength(NullRec) = 0;

RecLength1(Record A, Record B):
  RecLength(ConcatRec(A,B)) = RecLength(A) + RecLength(B);

RecLength2(char X):
  RecLength(RecofChar(X)) =
    if X <> NullChar
      then 1
      else 0;


IsNull0:
  IsNull(NullRec) = true;

IsNull1(Record A, Record B):
  IsNull(ConcatRec(A,B)) = IsNull(A) .and. IsNull(B);

IsNull2(char X):
  IsNull(RecofChar(X)) = X = NullChar;


First0:
  First(NullRec) = NullRec;

First1(Record A, Record B):
  First(ConcatRec(A,B)) =
    if IsNull(A)
      then First(B)
      else First(A);

First2(char X):
  First(RecofChar(X)) = RecofChar(X);


CharofRec0:
  CharofRec(NullRec) = NullChar;

CharofRec1(Record A, Record B):
  CharofRec(ConcatRec(A,B)) =
    if IsNull(A)
      then CharofRec(B)
      else CharofRec(A);
```

Appendix IV - Specifications from the case study

```
CharofRec2(char X):
  CharofRec(RecofChar(X)) = X;


Rest0:
  Rest(NullRec) = NullRec;

Rest1(Record A, Record B):
  Rest(ConcatRec(A,B)) =
    if IsNull(A)
      then Rest(B)
      else ConcatRec(Rest(A),B);

Rest2(char X):
  Rest(RecofChar(X)) = NullRec;


RecordEqual1(Record A, Record B):
  RecordEqual(A,B) =
    if IsNull(A)
      then IsNull(B)
      else if IsNull(B)
        then false
        else CharofRec(A) = CharofRec(B) .and.
             RecordEqual(Rest(A),Rest(B));


RecofString1(string A):
  RecofString(A) =
    if A = ''
      then NullRec
      else if Length(A) > 1
        then
          ConcatRec(RecofChar(charf(A)),RecofString(A[2]))
        else RecofChar(charf(A));


StringofRec1(Record X):
  StringofRec(X) =
    if IsNull(X)
      then ''
      else CharofRec(X) .con. StringofRec(Rest(X));


SubRec1(Record A, int X, int Y):
  SubRec(A,X,Y) =
    if X > 1
      then SubRec(Rest(A),X-1,Y)
      else if Y > 0
        then ConcatRec(First(A),SubRec(Rest(A),X,Y-1))
        else NullRec;
```

Appendix IV - Specifications from the case study

```
MatchRec1(Record A, Record B):
  MatchRec(A,B) =
    if RecLength(A) < RecLength(B) .or. IsNull(B)
      then 0
      else if RecordEqual(SubRec(A,1,RecLength(B)),B)
        then 1
        else if MatchRec(Rest(A),B) <> 0
          then 1 + MatchRec(Rest(A),B)
          else 0;


EscapedFind1(Record X, char A, char B):
  EscapedFind(X,A,B) =
    if IsNull(X)
      then 0
      else if CharofRec(X) = A
        then 1
        else if CharofRec(X) = B
          then if EscapedFind(SubRec(X,3,1000),A,B) <> 0
            then 2 + EscapedFind(SubRec(X,3,1000),A,B)
            else 0
          else if EscapedFind(SubRec(X,2,1000),A,B) <> 0
            then 1 + EscapedFind(SubRec(X,2,1000),A,B)
            else 0;


SortedCharSet1(char C, Record X):
  SortedCharSet(C,X) =
    if C = NullChar
      then X
      else if IsNull(X)
        then RecofChar(C)
        else if CharofRec(X) < C
          then
            ConcatRec(First(X),SortedCharSet(C,Rest(X)))
          else if CharofRec(X) = C
            then X /* don't duplicate! */
            else ConcatRec(RecofChar(C),X);


IntvalofRec1(Record A):
  IntvalofRec(A) = IntvalHelp(A,0);

IntvalHelp1(Record A, int I):
  IntvalHelp(A,I) =
    if IsNull(A) .or.
       (MatchRec(RecofString('0123456789'),First(A)) = 0)
      then I
      else
        IntvalHelp(Rest(A),
          I*10+MatchRec(RecofString('123456789'),First(A)));
```

Appendix IV - Specifications from the case study

```
FirstisDigit1(Record R):
  FirstisDigit(R) =
    if IsNull(R)
      then false
      else
        (CharofRec(R) >= "0" .and. CharofRec(R) <= "9");


FirstisA1(char A, Record R):
          /* reverse parameters to use existing test sets */
  FirstisA(R,A) =
    if IsNull(R)
      then false
      else CharofRec(R) = A;
```

Appendix IV - Specifications from the case study

11.2. Axioms for Files

```
Attop0:
  Attop(NewFile) = true;

Attop1(Record R, Files X):
  Attop(Insert(R,X)) = false;

Attop2(Files X):
       /* Delete the record backed over! */
  Attop(Back(X)) = Attop(Delete(X));

Atend0:
  Atend(NewFile) = true;

Atend1(Record R, Files X):
  Atend(Insert(R,X)) = Atend(X);

Atend2(Files X):
  Atend(Back(X)) = IsEmpty(X);

CurrentRec0:
  CurrentRec(NewFile) = NullRec;

CurrentRec1(Record R, Files X):
  CurrentRec(Insert(R,X)) = R;

CurrentRec2(Files X):
  CurrentRec(Back(X)) =
    if Attop(Back(X))
      then NullRec
      else CurrentRec(Delete(X)); /* get rid of a rec */

CurrentAddress0:
  CurrentAddress(NewFile) = 0;

CurrentAddress1(Record R, Files X):
  CurrentAddress(Insert(R,X)) = CurrentAddress(X) + 1;

CurrentAddress2(Files X):
  CurrentAddress(Back(X)) = if Attop(X)
    then 0
    else CurrentAddress(X) - 1;

FirstRec0:
  FirstRec(NewFile) = NullRec;

FirstRec1(Record R, Files X):
  FirstRec(Insert(R,X)) =
    if Attop(X)
      then R
      else FirstRec(X);

FirstRec2(Files X):
  FirstRec(Back(X)) = FirstRec(X);

FilesLength0:
  FilesLength(NewFile) = 0;

FilesLength1(Record R, Files X):
  FilesLength(Insert(R,X)) = 1 + Fileslength(X);

FilesLength2(Files X):
  FilesLength(Back(X)) = Fileslength(X);

IsEmpty0:
  IsEmpty(NewFile) = true;

IsEmpty1(Record R, Files X):
  IsEmpty(Insert(R,X)) = false;

IsEmpty2(Files X):
  IsEmpty(Back(X)) = IsEmpty(X);

Advance0:
  Advance(NewFile) = NewFile;

Advance1(Files X):
  Advance(Back(X)) = if Attop(X)
    then Advance(X)
    else X;

Advance2(Record R, Files X):
  Advance(Insert(R,X)) = if Atend(X)
    then Insert(R,X)
    else Insert(CurrentRec(Advance(X)),
      Replace(R,Advance(X)));

FilesEqual1(Files X, Files Y):
  FilesEqual(X,Y) =
    if Attop(X)
      then if Attop(Y)
        then if IsEmpty(X)
          then IsEmpty(Y)
          else if IsEmpty(Y)
            then false
            else FilesEqual(Advance(X),Advance(Y))
        else false
      else if Attop(Y)
        then false
        else if RecordEqual(CurrentRec(X),CurrentRec(Y))
          then FilesEqual(Delete(X),Delete(Y))
          else false;

Replace0(Record R):
  Replace(R,NewFile) = NewFile;

Replace1(Record R1, Record R2, Files X):
  Replace(R1,Insert(R2,X)) = Insert(R1,X);

Replace2(Record R, Files X):
  Replace(R,Back(X)) =
    if Attop(Back(X))
      then Back(X)
      else
        Back(Insert(CurrentRec(X),Replace(R,Delete(X))));

Delete0:
  Delete(NewFile) = NewFile;

Delete1(Record R, Files X):
  Delete(Insert(R,X)) = X;

Delete2(Files X):
  Delete(Back(X)) =
    if Attop(Back(X))
      then Back(X)
      else Back(Insert(CurrentRec(X),Delete(Delete(X))));

Nth1(Files X, int N):
  Nth(X,N) =
    if N <= 0
      then X
      else Nth(Advance(X),N-1);


Goto1(Files X, int N):
  Goto(X,N) = Nth(TopofFile(X),N);


TopofFile1(Files X):
  TopofFile(X) =
    if Attop(X)
      then X
      else TopofFile(Back(X));
```

11.3. Axioms for Patterns

```
/* First, five million silly axioms to make the rest of them
 * intelligible. These are all boring, and can be skipped on
 * first and all subsequent readings except for the
 * "IsChoice3" axiom, which explains (HA!) what a legal
 * choice pattern looks like.
 */


IsArb1:
  IsArb(PrintPattern(´´,Arb)) =
    true; /* print patterns to get it used! */

IsArb2:
  IsArb(PrintPattern(´´,Bol)) =
    false; /* print patterns to get it used! */

IsArb3(Record R):
  IsArb(Choice(R)) = false;

IsArb4:
  IsArb(Closure) = false;

IsArb5:
  IsArb(Eol) = false;

IsArb6:
  IsArb(ErrorPat) = false;

IsArb7(char C):
  IsArb(Lit(C)) = false;

IsArb8:
  IsArb(NullPat) = false;

IsArb9(Pattern A, Pattern B):
  IsArb(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsArb(B)) .or.
    (IsArb(A) .and. IsNullPat(B));


IsBol1:
  IsBol(Arb) = false;

IsBol2:
  IsBol(Bol) = true;

IsBol3(Record R):
  IsBol(Choice(R)) = false;
```

Appendix IV - Specifications from the case study

```
IsBol4:
  IsBol(Closure) = false;

IsBol5:
  IsBol(Eol) = false;

IsBol6:
  IsBol(ErrorPat) = false;

IsBol7(char C):
  IsBol(Lit(C)) = false;

IsBol8:
  IsBol(NullPat) = false;

IsBol9(Pattern A, Pattern B):
  IsBol(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsBol(B)) .or.
    (IsBol(A) .and. IsNullPat(B));


IsChoice1:
  IsChoice(Arb) = false;

IsChoice2:
  IsChoice(Bol) = false;

IsChoice3(Record R):
  IsChoice(Choice(R)) =
    if (CharofRec(R) <> "[") .or.
       MatchRec(RecofString('^-'),First(Rest(R))) .or.
       (EscapedFind(R,"]","@") <> RecLength(R))
     then false /* its ends are a mess */
     else /* it ends and start alright, check middle */
       if CharofRec(Rest(R)) = "@"
        then /* an escaped char to lead off! */
       if RecLength(R) = 4
        then true /* just the escaped char present */
        else /* need to check the rest of the choices */
          IsChoice(Choice(ConcatRec(RecofChar("["),
            SubRec(R,4,1000))))
        else /* not an escape char to lead off... */
       if CharofRec(SubRec(R,3,1)) = "-"
        then /* check for a legal range */
          if RecLength(R) < 5 .or.
            CharofRec(SubRec(R,4,1)) <= CharofRec(Rest(R))
              .or. .not.
             ((CharofRec(SubRec(R,4,1)) <= "9" .and.
               "0" <= CharofRec(Rest(R))) .or.
             (CharofRec(SubRec(R,4,1)) <= "Z" .and.
               "A" <= CharofRec(Rest(R))) .or.
```

```
                (CharofRec(SubRec(R,4,1)) <= "z" .and.
                  "a" <= CharofRec(Rest(R))))
                then false /* not a legal range! */
                else /* check other choices if present */
                  if RecLength(R) = 5
                    then true
                    else /* check other choices */
                      IsChoice(Choice(ConcatRec(
                        RecofChar("["),SubRec(R,5,1000))))
            else /* not a range! it starts legally! */
              if RecLength(R) = 3
                then true /* and ends here! */
                else /* need to check other choices! */
                  IsChoice(Choice(ConcatRec(RecofChar("["),
                    SubRec(R,3,1000))));

IsChoice4:
  IsChoice(Closure) = false;

IsChoice5:
  IsChoice(Eol) = false;

IsChoice6:
  IsChoice(ErrorPat) = false;

IsChoice7(char C):
  IsChoice(Lit(C)) = false;

IsChoice8:
  IsChoice(NullPat) = false;

IsChoice9(Pattern A, Pattern B):
  IsChoice(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsChoice(B)) .or.
    (IsChoice(A) .and. IsNullPat(B));


ValofChoice1:
  ValofChoice(Arb) = NullRec;

ValofChoice2:
  ValofChoice(Bol) = NullRec;

ValofChoice3(Record R):
  ValofChoice(Choice(R)) =
if IsNegChoice(Choice(R))
  then
    ValofChoice(Choice(ConcatRec(RecofChar("["),
      Rest(Rest(R)))))
else if .not. IsChoice(Choice(R)) /* check choice! */
  then NullRec /* Take care of the easy ones */
```

```
else if CharofRec(Rest(R)) = "@"
    /* need to build up the list of the letters */
  then /* an escaped char to lead off! */
    if RecLength(R) = 4
      then /* just the escaped char present */
        SubRec(R,3,1)
      else /* need to list the rest of the choices */
        SortedCharSet(CharofRec(SubRec(R,3,1)),
          ValofChoice(Choice(ConcatRec(RecofChar("["),
            SubRec(R,4,1000)))))
else if CharofRec(SubRec(R,3,1)) = "-"
    /* not an escape char to lead off... */
  then /* expand the legal range */
    if intval(CharofRec(Rest(R))) + 1 =
        intval(CharofRec(SubRec(R,4,1)))
      then /* these two chars are the ends of the range */
        if RecLength(R) = 5
          then RecofString(CharofRec(Rest(R)) .con.
            CharofRec(SubRec(R,4,1)))
          else
            SortedCharSet(CharofRec(Rest(R)),
              SortedCharSet(CharofRec(SubRec(R,4,1)),
                ValofChoice(Choice(ConcatRec(
                  RecofChar("["),SubRec(R,5,1000))))))
      else /* not at the ends of the range */
        SortedCharSet(CharofRec(Rest(R)),
          ValofChoice(Choice(ConcatRec(RecofChar("["),
            ConcatRec(RecofChar(charval(1+
              intval(CharofRec(Rest(R))))),
                SubRec(R,3,1000))))))
else if RecLength(R) = 3 /* it starts legally! */
  then SubRec(R,2,1) /* and ends here! */
else /* need to check other choices! */
  SortedCharSet(CharofRec(Rest(R)),
    ValofChoice(Choice(ConcatRec(RecofChar("["),
      SubRec(R,3,1000)))));

ValofChoice4:
  ValofChoice(Closure) = NullRec;

ValofChoice5:
  ValofChoice(Eol) = NullRec;

ValofChoice6:
  ValofChoice(ErrorPat) = NullRec;

ValofChoice7(char C):
  ValofChoice(Lit(C)) = NullRec;

ValofChoice8:
  ValofChoice(NullPat) = NullRec;
```

```
ValofChoice9(Pattern A, Pattern B):
  ValofChoice(ConcPats(A,B)) =
    if IsNullPat(A)
     then ValofChoice(B)
     else if IsNullPat(B)
       then ValofChoice(A)
       else NullRec;


IsNegChoice1:
  IsNegChoice(Arb) = false;

IsNegChoice2:
  IsNegChoice(Bol) = false;

IsNegChoice3(Record R):
  IsNegChoice(Choice(R)) =
    CharofRec(Rest(R)) = "^" .and.
    IsChoice(Choice(ConcatRec(SubRec(R,1,1),
      SubRec(R,3,1000))));

IsNegChoice4:
  IsNegChoice(Closure) = false;

IsNegChoice5:
  IsNegChoice(Eol) = false;

IsNegChoice6:
  IsNegChoice(ErrorPat) = false;

IsNegChoice7(char C):
  IsNegChoice(Lit(C)) = false;

IsNegChoice8:
  IsNegChoice(NullPat) = false;

IsNegChoice9(Pattern A, Pattern B):
  IsNegChoice(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsNegChoice(B)) .or.
    (IsNegChoice(A) .and. IsNullPat(B));


IsClosure1:
  IsClosure(Arb) = false;

IsClosure2:
  IsClosure(Bol) = false;

IsClosure3(Record R):
  IsClosure(Choice(R)) = false;
```

```
IsClosure4:
  IsClosure(Closure) = true;

IsClosure5:
  IsClosure(Eol) = false;

IsClosure6:
  IsClosure(ErrorPat) = false;

IsClosure7(char C):
  IsClosure(Lit(C)) = false;

IsClosure8:
  IsClosure(NullPat) = false;

IsClosure9(Pattern A, Pattern B):
  IsClosure(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsClosure(B)) .or.
    (IsClosure(A) .and. IsNullPat(B));


IsEol1:
  IsEol(Arb) = false;

IsEol2:
  IsEol(Bol) = false;

IsEol3(Record R):
  IsEol(Choice(R)) = false;

IsEol4:
  IsEol(Closure) = false;

IsEol5:
  IsEol(Eol) = true;

IsEol6:
  IsEol(ErrorPat) = false;

IsEol7(char C):
  IsEol(Lit(C)) = false;

IsEol8:
  IsEol(NullPat) = false;

IsEol9(Pattern A, Pattern B):
  IsEol(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsEol(B)) .or.
    (IsEol(A) .and. IsNullPat(B));
```

Appendix IV - Specifications from the case study

```
IsErrorPat1:
  IsErrorPat(Arb) = false;

IsErrorPat2:
  IsErrorPat(Bol) = false;

IsErrorPat3(Record R):
  IsErrorPat(Choice(R)) =
    (.not. IsChoice(Choice(R))) .and.
    (.not. IsNegChoice(Choice(R)));

IsErrorPat4:
  IsErrorPat(Closure) = false;

IsErrorPat5:
  IsErrorPat(Eol) = false;

IsErrorPat6:
  IsErrorPat(ErrorPat) = true;

IsErrorPat7(char C):
  IsErrorPat(Lit(C)) = false;

IsErrorPat8:
  IsErrorPat(NullPat) = false;

IsErrorPat9(Pattern A, Pattern B):
  IsErrorPat(ConcPats(A,B)) =
    (IsErrorPat(A) .or. IsErrorPat(B));


IsLit1:
  IsLit(Arb) = false;

IsLit2:
  IsLit(Bol) = false;

IsLit3(Record R):
  IsLit(Choice(R)) = false;

IsLit4:
  IsLit(Closure) = false;

IsLit5:
  IsLit(Eol) = false;

IsLit6:
  IsLit(ErrorPat) = false;

IsLit7(char C):
  IsLit(Lit(C)) = true;
```

Appendix IV - Specifications from the case study

```
IsLit8:
  IsLit(NullPat) = false;

IsLit9(Pattern A, Pattern B):
  IsLit(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsLit(B)) .or.
    (IsLit(A) .and. IsNullPat(B));


ValofLit1:
  ValofLit(Arb) = NullRec;

ValofLit2:
  ValofLit(Bol) = NullRec;

ValofLit3(Record R):
  ValofLit(Choice(R)) = NullRec;

ValofLit4:
  ValofLit(Closure) = NullRec;

ValofLit5:
  ValofLit(Eol) = NullRec;

ValofLit6:
  ValofLit(ErrorPat) = NullRec;

ValofLit7(char C):
  ValofLit(Lit(C)) = RecofChar(C);

ValofLit8:
  ValofLit(NullPat) = NullRec;

ValofLit9(Pattern A, Pattern B):
  ValofLit(ConcPats(A,B)) =
    if IsNullPat(A)
      then ValofLit(B)
      else if IsNullPat(B)
        then ValofLit(A)
        else NullRec;


IsNullPat1:
  IsNullPat(Arb) = false;

IsNullPat2:
  IsNullPat(Bol) = false;

IsNullPat3(Record R):
  IsNullPat(Choice(R)) = false;
```

```
IsNullPat4:
  IsNullPat(Closure) = false;

IsNullPat5:
  IsNullPat(Eol) = false;

IsNullPat6:
  IsNullPat(ErrorPat) = false;

IsNullPat7(char C):
  IsNullPat(Lit(C)) = false;

IsNullPat8:
  IsNullPat(NullPat) = true;

IsNullPat9(Pattern A, Pattern B):
  IsNullPat(ConcPats(A,B)) =
    (IsNullPat(A) .and. IsNullPat(B));


FirstPat1:
  FirstPat(Arb) = Arb;

FirstPat2:
  FirstPat(Bol) = Bol;

FirstPat3(Record R):
  FirstPat(Choice(R)) = Choice(R);

FirstPat4:
  FirstPat(Closure) = Closure;

FirstPat5:
  FirstPat(Eol) = Eol;

FirstPat6:
  FirstPat(ErrorPat) = ErrorPat;

FirstPat7(char C):
  FirstPat(Lit(C)) = Lit(C);

FirstPat8:
  FirstPat(NullPat) = NullPat;

FirstPat9(Pattern A, Pattern B):
  FirstPat(ConcPats(A,B)) =
    if IsErrorPat(A) .or. IsErrorPat(B)
      then ErrorPat
      else if IsNullPat(A)
        then FirstPat(B)
        else FirstPat(A);
```

Appendix IV - Specifications from the case study

```
LastPat1:
  LastPat(Arb) = Arb;

LastPat2:
  LastPat(Bol) = Bol;

LastPat3(Record R):
  LastPat(Choice(R)) = Choice(R);

LastPat4:
  LastPat(Closure) = Closure;

LastPat5:
  LastPat(Eol) = Eol;

LastPat6:
  LastPat(ErrorPat) = ErrorPat;

LastPat7(char C):
  LastPat(Lit(C)) = Lit(C);

LastPat8:
  LastPat(NullPat) = NullPat;

LastPat9(Pattern A, Pattern B):
  LastPat(ConcPats(A,B)) =
    if IsErrorPat(A) .or. IsErrorPat(B)
      then ErrorPat
      else if IsNullPat(B)
        then LastPat(A)
        else LastPat(B);


RestofPat1:
  RestofPat(Arb) = NullPat;

RestofPat2:
  RestofPat(Bol) = NullPat;

RestofPat3(Record R):
  RestofPat(Choice(R)) =
    if IsErrorPat(Choice(R))
      then ErrorPat
      else NullPat;

RestofPat4:
  RestofPat(Closure) = NullPat;

RestofPat5:
  RestofPat(Eol) = NullPat;
```

```
RestofPat6:
  RestofPat(ErrorPat) = ErrorPat;

RestofPat7(char C):
  RestofPat(Lit(C)) = NullPat;

RestofPat8:
  RestofPat(NullPat) = NullPat;

RestofPat9(Pattern A, Pattern B):
  RestofPat(ConcPats(A,B)) =
    if IsErrorPat(A) .or. IsErrorPat(B)
      then ErrorPat
      else if IsNullPat(A)
        then RestofPat(B)
        else ConcPats(RestofPat(A),B);


AllButLastPat1:
  AllButLastPat(Arb) = NullPat;

AllButLastPat2:
  AllButLastPat(Bol) = NullPat;

AllButLastPat3(Record R):
  AllButLastPat(Choice(R)) =
    if IsErrorPat(Choice(R))
      then ErrorPat
      else NullPat;

AllButLastPat4:
  AllButLastPat(Closure) = NullPat;

AllButLastPat5:
  AllButLastPat(Eol) = NullPat;

AllButLastPat6:
  AllButLastPat(ErrorPat) = ErrorPat;

AllButLastPat7(char C):
  AllButLastPat(Lit(C)) = NullPat;

AllButLastPat8:
  AllButLastPat(NullPat) = NullPat;

AllButLastPat9(Pattern A, Pattern B):
  AllButLastPat(ConcPats(A,B)) =
    if IsErrorPat(A) .or. IsErrorPat(B)
      then ErrorPat
      else if IsNullPat(B)
        then AllButLastPat(A)
      else ConcPats(A,AllButLastPat(B));

ConcPats1(Pattern P):
  ConcPats(NullPat,P) = P;

ConcPats2(Pattern P):
  ConcPats(P,NullPat) = P;

ConcPats3(Pattern P):
  ConcPats(ErrorPat,P) = ErrorPat;

ConcPats4(Pattern P):
  ConcPats(P,ErrorPat) = ErrorPat;
/+ eject +/
/* Now, these are the really interesting axioms! */


PatfromRec1(Record R):
  PatfromRec(R) =
    if RecordEqual(R,NullRec)
      then NullPat
    else /* there's something there! */
      if CharofRec(R) = "@"
        then
          if RecLength(R) = 1
            then ErrorPat /* no escaped char present */
            else ConcPats(Lit(CharofRec(Rest(R))),
              PatfromRec(SubRec(R,3,1000)))
    else /* not a literal! */
      if CharofRec(R) = "?"
        then ConcPats(Arb,PatfromRec(Rest(R)))
    else /* not an arb! */
      if CharofRec(R) = "*"
        then ConcPats(Closure,
          PatfromRec(Rest(R)))
    else /* not a closure! */
      if CharofRec(R) = "^"
        then ConcPats(Bol,
          PatfromRec(Rest(R)))
    else /* not a bol! */
      if CharofRec(R) = "$"
        then
          ConcPats(Eol,PatfromRec(Rest(R)))
    else /* not eol! */
      if CharofRec(R) = "["
        then /* try a choice! */
          ConcPats(Choice(SubRec(R,1,
            EscapedFind(R,"]","@"))),
              PatfromRec(SubRec(R,
                EscapedFind(R,"]","@")+1,1000)))
    else /* isn't a choice */
      ConcPats(Lit(CharofRec(R)),PatFromRec(Rest(R)));


PatternEquall(Pattern A, Pattern B):
  PatternEqual(A,B) =
if IsNullPat(A)
  then IsNullPat(B)
else if IsErrorPat(A)
  then IsErrorPat(B)
else if IsNullPat(B) .or. IsErrorPat(B)
  then false
else if .not. IsNullPat(RestofPat(A)) .or.
     .not. IsNullPat(RestofPat(B))
  then /* not both primitive patterns, try again! */
    PatternEqual(FirstPat(A),FirstPat(B)) .and.
      PatternEqual(RestofPat(A),RestofPat(B))
else if IsArb(A) /* we have primitive patterns! */
  then IsArb(B)
else if IsBol(A)
  then IsBol(B)
else if IsChoice(A)
  then IsChoice(B) .and.
    RecordEqual(ValofChoice(A),ValofChoice(B))
else if IsClosure(A)
  then IsClosure(B)
else if IsEol(A)
  then IsEol(B)
else if IsLit(A)
  then IsLit(B) .and.
    RecordEqual(ValofLit(A),ValofLit(B))
else
/* 'A' MUST BE a "neg choice"! The next line is unneeded:
 * IsNegChoice(A) .and.
 */
  IsNegChoice(B) .and.
    RecordEqual(ValofChoice(A),ValofChoice(B));


MatchPatl(Record R, Pattern P):
  MatchPat(R,P) =
if IsNullPat(P) /* Is this a Null pattern */
  then 1 /* a Null pattern always matches */
else if IsErrorPat(P) /* is this an error pattern */
  then 0 /* error patterns never match */
else if IsEol(FirstPat(P)) /* is this end of line */
  then /* End of Line pattern */
    if IsNull(R) .and. IsNullPat(RestofPat(P))
      then /* we're at end of the record and pattern! */
        1
      else /* try sliding over! */
```

```
          if MatchPat(Rest(R),P) <> 0
               /* if it matches, P = Eol! */
            then MatchPat(Rest(R),Eol) + 1 /* slide! */
            else 0 /* can't slide either! */
  else if IsBol(FirstPat(P)) /* is this a BOL */
    then /* this pattern may not float across the line! */
      if (.not. IsBol(FirstPat(RestofPat(P)))) .and.
          MatchPat(R,RestofPat(P)) = 1
        then 1
        else 0
  else if IsClosure(FirstPat(P)) /* is it closure */
    then 0 /* pattern starts with closure, an error! */
  else if IsArb(FirstPat(P)) /* is it an Arb */
    then /* this is an Arb Pattern! */
      if IsClosure(FirstPat(RestofPat(P)))
        then /* need to handle closures! */
          if IsBol(FirstPat(RestofPat(RestofPat(P))))
            then 0 /* Can't match next Bol */
            else /* no Bols hanging around! */
              if IsNull(R) .or. MatchPat(Rest(R),P) = 0
                then /* Dont skip over the current char! */
                  MatchPat(R,RestofPat(RestofPat(P)))
                else 1 /* matched later, arbs start here */
        else /* no closure hanging around! */
          if IsBol(FirstPat(RestofPat(P))) .or. IsNull(R)
              .or. MatchPat(Rest(R),RestofPat(P)) = 0
            then 0 /* can't match off any chars or float! */
            else /* o.k., see where the rest floats to */
              MatchPat(Rest(R),RestofPat(P))
  else if IsLit(FirstPat(P)) /* is this a Lit */
    then /* this is an Lit Pattern! */
      if IsClosure(FirstPat(RestofPat(P)))
        then /* need to handle closures! */
          if IsBol(FirstPat(RestofPat(RestofPat(P))))
            then 0 /* Can't match next Bol */
            else /* no Bols hanging around! */
              if IsNull(R) .or. MatchPat(Rest(R),P) = 0
                then /* no chance of matching further on! */
                        /* try matching zero chars */
                  MatchPat(R,RestofPat(RestofPat(P)))
                else /* see where the match worked */
                  if (CharofRec(ValofLit(FirstPat(P)))<>
                         CharofRec(R) .and.
                         MatchPat(R,RestofPat(RestofPat(P)))=1)
                       .or. (MatchPat(Rest(R),P) = 1 .and.
                         CharofRec(ValofLit(FirstPat(P)))=
                         CharofRec(R))
                    then 1 /* did not slide */
                    else /* did slide! It matchs later */
                      MatchPat(Rest(R),P) + 1
        else /* no closure hanging around! */
        if IsNull(R) .or. IsBol(FirstPat(RestofPat(P)))
          then 0 /* can't match off any chars or float! */
          else /* o.k., check the first char and then,
                    the rest of the pattern */
            if CharofRec(ValofLit(FirstPat(P)))=
                  CharofRec(R) .and.
                 MatchPat(Rest(R),RestofPat(P)) = 1
              then 1 /* we match starting here! */
              else /* try floating across */
                if MatchPat(Rest(R),P) <> 0
                  then MatchPat(Rest(R),P) + 1
                  else 0
/* Only legal patterns can be created, so assume:
 * else
 *    if IsNegChoice(FirstPat(P)) .or.
 *         IsChoice(FirstPat(P)) /* choice */
 *      then /* this is an choice Pattern! */
 */
else if IsClosure(FirstPat(RestofPat(P)))
  then /* need to handle closures! */
    if IsBol(FirstPat(RestofPat(RestofPat(P))))
      then 0 /* Can't match next Bol */
      else /* no Bols hanging around! */
        if (.not. IsNull(R) .and.
             ((MatchRec(ValofChoice(FirstPat(P)),
               First(R))<>0) =
             IsChoice(FirstPat(P))) .and.
             MatchPat(Rest(R),P) = 1) .or.
             MatchPat(R,RestofPat(RestofPat(P))) = 1
          then 1 /* Match starts here! */
          else /* Current char is not o.k., see if it
                    matches later on */
            if MatchPat(Rest(R),P) <> 0
              then /* It does! */
                MatchPat(Rest(R),P) + 1
              else /* It does not! */
                0
  else /* no closure hanging around! */
    if IsNull(R) .or. IsBol(FirstPat(RestofPat(P)))
      then 0 /* can't match off any chars or float! */
      else /* o.k., drop the first char and pattern */
        if MatchPat(Rest(R),RestofPat(P)) = 1 .and.
             (MatchRec(ValofChoice(FirstPat(P)),
               First(R)) <> 0) = IsChoice(FirstPat(P))
          then 1 /* we match starting here! */
          else /* try floating across */
            if MatchPat(Rest(R),P) <> 0
              then MatchPat(Rest(R),P) + 1
              else 0
/* not needed, because ALL patterns ARE viable!
 *  else 0 /* not a viable pattern */
 */
 ; /* end of axiom! */


MatchLenl(Record R, Pattern P):
  MatchLen(R,P) =
if IsNull(R) .or.
    IsNullPat(P) /* Null records and Null patterns */
  then 0 /* always have nothing left to match! */
else if IsErrorPat(P) /* is this an error pattern */
  then 0 /* error patterns never match */
else if IsEol(FirstPat(P)) /* is this end of line */
  then 0 /* EOL pattern allows nothing to follow */
else if IsBol(FirstPat(P)) /* is this a BOL */
  then /* this pattern may not float across the line! */
    if (.not. IsBol(FirstPat(RestofPat(P))))
         .and. MatchPat(R,RestofPat(P)) = 1
      then MatchLen(R,RestofPat(P)) /*Match starts here!*/
      else 0
else if IsClosure(FirstPat(P)) /* is it closure */
  then 0 /* pattern starts with closure, an error! */
else if IsArb(FirstPat(P)) /* is it an Arb */
  then /* this is an Arb Pattern! */
    if IsClosure(FirstPat(RestofPat(P)))
      then /* need to handle closures! */
        if IsBol(FirstPat(RestofPat(RestofPat(P))))
          then 0 /* Can't match next Bol */
          else /* no Bols hanging around! */
            if MatchPat(Rest(R),P) = 0
              then /* Dont skip over the current char! */
                MatchLen(R,RestofPat(RestofPat(P)))
              else
                MatchLen(Rest(R),P) + 1 /* match here */
      else /* no closure hanging around! */
        if IsBol(FirstPat(RestofPat(P)))
          .or. MatchPat(Rest(R),RestofPat(P)) = 0
          then 0 /* can't match off any chars or float! */
          else /* o.k., counting the current char */
            MatchLen(Rest(R),RestofPat(P)) + 1
else if IsLit(FirstPat(P)) /* is this a Lit */
  then /* this is an Lit Pattern! */
    if IsClosure(FirstPat(RestofPat(P)))
      then /* need to handle closures! */
        if IsBol(FirstPat(RestofPat(RestofPat(P))))
          then 0 /* Can't match next Bol */
          else /* no Bols hanging around! */
            if MatchPat(Rest(R),P) = 0
              then /* no chance of matching further on! */
                     /* try matching zero chars */
                MatchLen(R,RestofPat(RestofPat(P)))
              else /* see where the match worked */
                if CharofRec(ValofLit(FirstPat(P)))<>
                    CharofRec(R) .and.
                    MatchPat(R,RestofPat(RestofPat(P)))=1
                  then /* stopped matching of the lits! */
                    MatchLen(R,RestofPat(RestofPat(P)))
                  else /* see if we start matching here */
                     if MatchPat(Rest(R),P) = 1 .and.
                         CharofRec(ValofLit(FirstPat(P))) =
                           CharofRec(R)
                       then /* didn't slide */
                         MatchLen(Rest(R),P)+1
                       else /* Didn't start match here! */
                         MatchLen(Rest(R),P)
      else /* no closure hanging around! */
          if IsBol(FirstPat(RestofPat(P)))
          then 0 /* can't match off any chars or float! */
          else /* o.k., check the first char and then,
                    the rest of the pattern */
            if CharofRec(ValofLit(FirstPat(P)))=
                  CharofRec(R) .and.
                MatchPat(Rest(R),RestofPat(P)) = 1
              then /* we match starting here! */
                MatchLen(Rest(R),RestofPat(P)) + 1
              else /* try floating across */
                if MatchPat(Rest(R),P) <> 0
                  then MatchLen(Rest(R),P)
                  else 0
/* only legal patterns can be created, so this is assumed:
 * else
 *   if IsNegChoice(FirstPat(P)) .or.
 *       IsChoice(FirstPat(P)) /* choice */
 *     then /* this is an choice Pattern! */
 */
else if IsClosure(FirstPat(RestofPat(P)))
  then /* need to handle closures! */
    if IsBol(FirstPat(RestofPat(RestofPat(P))))
      then 0 /* Can't match next Bol */
      else /* no Bols hanging around! */
        if ((MatchRec(ValofChoice(FirstPat(P)),
               First(R))<>0)=IsChoice(FirstPat(P))) .and.
            MatchPat(Rest(R),P) = 1
          then /* this char starts it! */
            MatchLen(Rest(R),P) + 1
          else /* Does not start at this char... */
            if MatchPat(R,RestofPat(RestofPat(P))) = 1
              then /* It really matchs starting here */
                MatchLen(R,RestofPat(RestofPat(P)))
              else /* Current char is not o.k., see if it
                        matches later on */
                 if MatchPat(Rest(R),P) <> 0
```

```
                    then /* It does! */
                      Matchlen(Rest(R),P)
                    else /* It does not! */
                      0
   else /* no closure hanging around! */
     if IsBol(FirstPat(RestofPat(P)))
       then 0 /* can't match off any chars or float! */
       else /* o.k., drop the first char and pattern */
         if MatchPat(Rest(R),RestofPat(P)) = 1 .and.
             (MatchRec(ValofChoice(FirstPat(P)),
               First(R)) <> 0) = IsChoice(FirstPat(P))
           then /* we match starting here! */
             MatchLen(Rest(R),RestofPat(P)) + 1
           else /* try floating across */
             MatchLen(Rest(R),P)
/* not needed, because ALL patterns ARE viable!
 *  else 0 /* not a viable pattern */
 */
 ; /* end of axiom! */
```

11.4. Axioms for the Editor

```
Editor1(Files Text, Files Commands):
  Editor(Text,Commands) =
if IsEmpty(Commands)
  then Text
else if .not. AtTop(Back(Commands))
  then Editor(Text,Delete(Back(Commands)))
else if AtTop(Commands)
  then /* need to point to first record! */
    Editor(Text,Advance(Commands))
else if FirstisA(CurrentRec(Commands),",")
  then /* Edit1Addr won't be able to detect the error! */
    Editor(Text,Delete(Commands)) /* ignore the error */
else /* first see if there is a target address */
  Editor(Edit1Addr(Text,
    SkipAddr(CurrentRec(Commands)),
      AddrfromCommand(Text,CurrentRec(Commands))),
        Advance(Delete(Commands)));



AddrfromCommand1(Files Text, Record Command):
  AddrfromCommand(Text,Command) =
if FirstisDigit(Command)
  then /* first address is line number */
    IntvalofRec(Command)
else if FirstisA(Command,".")
  then /* first address is the current line! */
    CurrentAddress(Text)
else if FirstisA(Command,"/")
  then /* first address is a search forward */
    if EscapedFind(Rest(Command),"/","@") <> 0
      then /* first address is from a search! */
        ForwardSearch(Text,
          PatfromRec(SubRec(Command,2,
            EscapedFind(Rest(Command),"/","@")-1)))
      else /* search specifier is messed up */
        CurrentAddress(Text)
else if FirstisA(Command,"
  then /* first address is a search backward */
    if EscapedFind(Rest(Command),"
      then /* first address is from a search! */
        BackwardSearch(Text,PatfromRec(SubRec(Command,2,
          EscapedFind(Rest(Command),"
      else /* search specifier is messed up */
        CurrentAddress(Text)
else if FirstisA(Command,"$")
  then /* first address is end of text file */
    FilesLength(Text)
else CurrentAddress(Text); /* fail safe */
```

Appendix IV - Specifications from the case study

```
SkipAddr1(Record Command):
  SkipAddr(Command) =
if FirstisDigit(Command)
  then /* address is line number */
    SubRec(Command,MatchPat(Command,
      PatfromRec(RecofString('[^0-9]'))),1000)
else if FirstisA(Command,".") .or.
    FirstisA(Command,"$")
  then /* address is the current line or end of file */
    Rest(Command)
else if FirstisA(Command,"/") .or.
    FirstisA(Command,"
  then /* address is a search */
    if EscapedFind(Rest(Command),
        CharofRec(Command),"@") <> 0
      then /* address is a search */
        SubRec(Command,EscapedFind(Rest(Command),
          CharofRec(Command),"@")+2,1000)
      else
        NullRec
else Command; /* No Address present??? */


EditlAddr1(Files Text, Record Command, int Address):
  EditlAddr(Text,Command,Address) =
    if FirstisA(Command,",")
      then /* need to peel off the next address */
        EditApply(Goto(Text,Address),
          SkipAddr(Rest(Command)),
           AddrfromCommand(Text,Rest(Command))-Address + 1)
      else EditApply(Goto(Text,Address),Command,1);


EditApply1(Files Text, Record Command, int Count):
  EditApply(Text,Command,Count) =
if Count <= 0
  then PrintCurrent(Text)
else if IsNull(Command) .or. FirstisA(Command,"p")
  then /* need to watch the count */
    if Count > 1
      then
        EditApply(Advance(PrintCurrent(Text)),
          Command,Count - 1)
      else PrintCurrent(Text)
else if FirstisA(Command,"d")
  then
    EditApply(Advance(Delete(Text)),RecofChar("d"),Count-1)
else if FirstisA(Command,"s")
  then
  if Escapedfind(Rest(Rest(Command)),
    CharofRec(Rest(Command)),"@") <> 0 .and.
    /* middle delimiter */
    EscapedFind(SubRec(Command,
     EscapedFind(Rest(Rest(Command)),
      CharofRec(Rest(Command)),"@")+3,
        /* was ´+1´, found via */
        /* expression monitoring */
       1000),CharofRec(Rest(Command)),"@")<>0
    /* ending delimiter */
   then /* both delimiters present */
    if Count > 1
     then
      EditApply(Advance(ApplySubstitute(Text,
       PatfromRec(SubRec(Command,3,
        EscapedFind(Rest(Rest(Command)),
         CharofRec(Rest(Command)),"@")-1)),
          SubRec(Command,
           EscapedFind(Rest(Rest(Command)),
            CharofRec(Rest(Command)),"@")+3,
             EscapedFind(SubRec(Command,
              EscapedFind(Rest(Rest(Command)),
               CharofRec(Rest(Command)),"@")
                +3,1000),
                 CharofRec(Rest(Command)),
                  "@")-1))),
                   Command,Count-1)
     else /* don´t advance! */
      PrintCurrent(ApplySubstitute(Text,PatfromRec(
       SubRec(Command,3,EscapedFind(Rest(Rest(
        Command)),CharofRec(Rest(Command)),"@")-1)),
         SubRec(Command,EscapedFind(Rest(Rest(
          Command)),
           CharofRec(Rest(Command)),"@")+3,
            EscapedFind(SubRec(Command,
             EscapedFind(Rest(Rest(Command)),
              CharofRec(Rest(Command)),"@")
               +3,1000),
                CharofRec(Rest(Command)),
                 "@")-1)))
   else /* missing delimeter??? */
     /* Apply no changes! */
    EditApply(Text,RecofString(´s///´),Count)
else if FirstisA(Command,"i") /* try for an insert */
 then /* It is insert! */
  if Count > 1
   then /* Advance between insertions! */
    EditApply(Advance(Insert(Rest(Command),Text)),
     Command,Count-1)
   else /* don´t advance the last time! */
    PrintCurrent(Insert(Rest(Command),Text))
```

Appendix IV - Specifications from the case study

```
else /* no editor command present??? */
  EditApply(Text,Command,0); /* at least cause the print! */


ApplySubstitutel(Files Text, Pattern P, Record R):
  ApplySubstitute(Text,P,R) =
    if IsErrorPat(P) .or.
        AtTop(Text) .or.
        MatchPat(CurrentRec(Text),P)=0
      then Text
      else
        Replace(ConCatRec(ConCatRec(
          SubRec(CurrentRec(Text),1,MatchPat(
            CurrentRec(Text),P)-1),R),SubRec(
              CurrentRec(Text),MatchPat(CurrentRec(Text),P)+
                MatchLen(CurrentRec(Text),P),1000)),Text);


BackwardSearchl(Files Text, Pattern P):
  BackwardSearch(Text,P) =
    if AtTop(Back(Text))
      then 0 /* NO WRAPAROUND! */
      else
        if MatchPat(CurrentRec(Back(Text)),P) <> 0
          then CurrentAddress(Back(Text))
          else BackwardSearch(Back(Text),P);


ForwardSearchl(Files Text, Pattern P):
  ForwardSearch(Text,P) =
    if AtEnd(Text)
      then FilesLength(Text)
      else
        if MatchPat(CurrentRec(Advance(Text)),P) <> 0
          then CurrentAddress(Advance(Text))
          else ForwardSearch(Advance(Text),P);
```